Amalgamation of Domain Specific Languages with Behaviour

Francisco Durán[a], Antonio Moreno-Delgado[a], Fernando Orejas[b], Steffen Zschaler[c]

[a]*Universidad de Málaga, Spain*
[b]*Universidad Politécnica de Cataluña, Spain*
[c]*King's College London, UK*

Abstract

Domain-specific languages (DSLs) become more useful the more specific they are to a particular domain. The resulting need for developing a substantial number of DSLs can only be satisfied if DSL development can be made as efficient as possible. One way in which to address this challenge is by enabling the reuse of (partial) DSLs in the construction of new DSLs. Reuse of DSLs builds on two foundations: a notion of DSL composition and theoretical results ensuring the safeness of composing DSLs with respect to the semantics of the component DSLs.

Given a graph-grammar formalisation of DSLs, in this paper, we build on graph transformation system morphisms to define parameterized DSLs and their instantiation by an amalgamation construction. Results on the protection of the behaviour along the induced morphisms allow us to safely reuse and combine definitions of DSLs to build more complex ones. We illustrate our proposal in e-Motions for a DSL for production-line systems and three independent DSLs for describing non-functional properties, namely response time, throughput, and failure rate.

1. Introduction

In Model-Driven Engineering (MDE) [53], models are used to specify, simulate, analyse, modify, and generate code. One of the key ingredients making this approach particularly attractive is the use of domain-specific languages (DSLs) [59] for the definition of such models. DSLs offer concepts specifically targeted at a particular domain, which allow experts in such domains to express their problems and requirements in their own languages. On the other hand, the higher amount of knowledge embedded in these concepts allows for much more complete and specialised generation of executable solution code from DSL models [35].

The application of these techniques to different domains has resulted in the proliferation of DSLs of very different nature: the more specific for a particular domain a DSL is, the more effective it is. However, DSLs are only viable if their development can be made efficient. With this goal in mind, DSLs are often defined by specifying their syntax in some standard formalisms, such as the Meta-Object Facility (MOF) [45], thus facilitating the use of generic frameworks for the management of models, including their composition, the definition of model transformations, etc.

Syntax is however just part of the story. Without a definition of the operational behaviour of the defined DSLs, we will not be able to simulate or analyse the defined models. In recent years, different formalisms have been proposed for the definition of the behaviour of DSLs, including UML behavioural models [21, 25], abstract state machines [12, 3], or in-place model transformations [11, 50]. Between all these approaches, we find the use of in-place model transformations particularly powerful, not only because of its expressiveness, but also because the use of transformations facilitates integration with other MDE environments and tools, such as simulators and model checkers.

While we have reasonably good knowledge of how to modularise DSL syntax, the modularisation of language semantics is an as yet unsolved issue. In our work, we build on a graph-grammar [6, 52, 16] formalisation of DSLs and on graph transformation system (GTS) morphisms to define composition operations on DSLs. Specifically, we define parameterized GTSs, that is, GTSs which have other GTSs as parameters. The instantiation of such parameterized GTSs is then provided by an amalgamation construction. Specifically, we are interested in how GTS morphisms preserve or protect behaviour, and what behaviour-related properties may be guaranteed on the morphisms induced by the amalgamation construction defining the instantiation of parameterized GTSs. These properties will be key, for instance, to be able to assert that the behaviour of a system has not changed when extended, if such an extension satisfies certain properties.

In this paper, we propose the use of parameterized DSLs, we present their implementation in the e-Motions system, and show its potential presenting the definition of the e-Motions implementation of a production-line DSL. This paper is an extension of our earlier work in [44, 13, 14]. Beyond a comprehensive presentation of the core concepts, this paper extends the work to situations where more than two DSLs need to be composed. We also provide more adequate notions of behaviour-aware morphisms. Specifically, we lift our definitions for behaviour-preserving, -reflecting, and -protecting morphisms to deal with traces, rather than only individual rules. In addition to extending formal results on these morphisms to these new notions, we provide formal results stating that composition of multiple GTSs is equivalent to iterative composition of individual GTSs, allowing us to directly apply previous results to the case of composing multiple DSLs. We present an implementation of these mechanisms as an extension of the e-Motions system, and show its use in our case study. Although we motivate and illustrate our approach using the e-Motions language [48], our proposal is language-independent, and all the results are presented for GTSs and adhesive HLR systems [40, 18].

The rest of the paper is structured as follows. Section 2 introduces behaviour-reflecting, -preserving and -protecting GTS morphisms, the construction of amalgamations in the category of GTSs and GTS morphisms, and several results on these amalgamations. Section 3 presents the e-Motions definition of a production-line DSL and a number of DSLs for non-functional properties. We then show how the composition operations presented in Section 2 can be used to derive a DSL for specifying and analysing non-functional properties of production-line systems. Section 4 presents our current implementation. The paper presents some related work in Section 5 and finishes with some conclusions and future work in Section 6.

2. Graph transformation and GTS amalgamations

Graph transformation [52, 16] is a formal, graphical and natural way of expressing graph manipulation based on rewriting rules. In graph-based modelling (and meta-modelling), graphs are used to define the static structures (e.g., classes and objects) that represent visual alphabets and sentences over them. In this section, we give a brief overview of the main elements of our approach to composing graph-transformation systems. Some of the results in this section builds on results presented in [13], appropriate references are provided when necessary. The main focus of our presentation in this paper is on the extension of these concepts to traces and to the composition of more than two graph-transformation systems.

We start the rest of this section by introducing adhesive categories in Section 2.1, and some basic concepts such as rules and rule morphisms in Section 2.2. This section also presents the constructions for rule amalgamation and multi-amalgamation. Section 2.3 focuses on typed graph transformation systems, and serves as ground on which to introduce, in Section 2.4, the key notions of GTS morphism and different types of GTS morphisms depending on the effects they have on the behaviour of the involved GTSs, and some results on them. Section 2.5 presents our construction for GTS amalgamation and key results on them related to the guarantees on behaviour of the induced morphisms. Section 2.6 extends the amalgamation construction to multiple GTSs.

2.1. Adhesive Categories

The original double-pushout (DPO) approach for graph transformation was defined for directed, labeled graphs. However, the fact that all constructions and results were based on categorical constructions led to a search for a characterisation of the kind of categories for which the fundamental theory of DPO graph transformation would apply. In this sense, Lack and Sobocinski defined in [40] the key notion of *Adhesive Category* that captured the essential properties that ensured the satisfaction of this fundamental theory. Thus, given proofs for adhesive categories of general results such as the Local Church-Rosser, or the Parallelism and Concurrency Theorem, they are automatically valid for any category which is proved an adhesive category.

Unfortunately, not all interesting categories of graph structures, like the category of attributed graphs, form adhesive categories. For this reason different variations on the notion of adhesivity have been proposed to cope with these cases. In particular, in this paper, we use the notion of *adhesive high-level replacement (HLR) category*, that was defined in [20] to cope with the case of attributed typed graphs. The concepts of adhesive and (weak) adhesive HLR categories abstract the foundations of a general class of models, and come together with a collection of general semantic techniques [40, 18]. The category of typed attributed graphs with inheritance, the one of interest to us, was proved to be adhesive HLR in [20].

Definition 1 (Adhesive HLR category [20]). *A category $\mathcal{C}$ together with a class of monomorphisms $\mathcal{M}$ is an* adhesive HLR category *if the following properties hold:*

1. *$\mathcal{M}$ is closed under isomorphism, composition and decomposition (i.e., if $g \circ f \in \mathcal{M}$ and $g \in \mathcal{M}$ then $f \in \mathcal{M}$).*

2. *$\mathcal{C}$ has pushouts and pullbacks along $\mathcal{M}$-morphisms. Moreover, $\mathcal{M}$-morphisms are closed under pushouts and pullbacks.*
3. *Pushouts in $\mathcal{C}$ along $\mathcal{M}$-morphisms are van Kampen squares, i.e. for any commutative diagram as the one below, assuming that h_1 and h_2 are $\mathcal{M}$-morphisms, if the bottom diagram is a pushout and the back faces are pullbacks then the top diagram is a pushout if and only if the front diagrams are pullbacks.*

$$\begin{tikzcd}
 & & A'_0 \arrow[dll, "h'_1"'] \arrow[dr, "h'_2"] \arrow[dd, "f_0"] & \\
A'_1 \arrow[dr, "g'_1"] \arrow[dd, "f_1"'] & & & A'_2 \arrow[dll, "g'_2"'] \arrow[dd, "f_2"] \\
 & A'_3 \arrow[dd, "f_3"] & A_0 \arrow[dll, "h_1"] \arrow[dr, "h_2"'] & \\
A_1 \arrow[dr, "g_1"'] & & & A_2 \arrow[dll, "g_2"] \\
 & A_3 & &
\end{tikzcd}$$

In the case of attributed typed graphs, $\mathcal{M}$ is the class of injective graph morphisms that are the identity when restricted to the attribute values. Intuitively, adhesive categories abstractly characterize the categories of set-like structures, where pushouts are some sort of union, pullbacks are some sort of intersection and pushout complements are some kind of set difference. Moreover, the van Kampen property ensures that we have some kind of distributivity between union and intersection.

2.2. Rules, rule morphisms, and rule amalgamations

In the DPO approach to graph transformation, a rule with application conditions p is of the form $(L \overset{l}{\leftarrow} K \overset{r}{\rightarrow} R, ac)$ with graphs L, K, and R, called, respectively, left-hand side, interface, and right-hand side, some kind of monomorphisms (typically, inclusions) l and r, and ac a *(nested) application condition* on L [31]. A graph transformation system (GTS) is a pair (P, π) where P is a set of rule names and π is a function mapping each rule name p into a rule $(L \overset{l}{\leftarrow} K \overset{r}{\rightarrow} R, ac)$.

The application of a transformation rule $p = (L \overset{l}{\leftarrow} K \overset{r}{\rightarrow} R, ac)$ to a graph G via a match $m : L \rightarrow G$, such that m satisfies ac, written $m \models ac$, is constructed as two gluings (1) and (2), which are pushouts in the corresponding graph category, leading to a direct transformation $G \overset{p,m}{\Longrightarrow} H$.

$$\begin{tikzcd}
ac & \triangleright & L \arrow[d, "m"'] \arrow[dr, phantom, "(1)"] & K \arrow[l, "l"'] \arrow[r, "r"] \arrow[d] \arrow[dr, phantom, "(2)"] & R \arrow[d] \\
 & & G & D \arrow[l] \arrow[r] & H
\end{tikzcd}$$

Note the use of the $\triangleright$ symbol to attach the application condition to its associated rule in the diagrams.

Application conditions may be positive or negative. Positive application conditions have the form $\exists a$, for a monomorphism $a : L \rightarrow C$, and demand a certain structure in addition to L. Negative application conditions of the form $\nexists a$ forbid such a structure. A match $m : L \to G$ satisfies a positive application condition $\exists a$ if there is a monomorphism $q : C \to G$ satisfying $q \circ a = m$. A match m satisfies a negative application condition $\nexists a$ if there is no such monomorphism. Given an application condition $\exists a$ or $\nexists a$, for a monomorphism $a : L \to C$, another application condition ac can be established on C, giving place to nested application conditions [31]. We can write Boolean expressions with them. Thus, given application conditions ac and ac', we may write $\neg ac$, $ac \wedge ac'$, $ac \vee ac'$, or $ac \Rightarrow ac'$ with the expected meaning. Given an application condition ac on L and a monomorphism $t : L \to L'$, then there is an application condition $\mathsf{Shift}(t, ac)$ on L' such that for all $m' : L' \to G$, $m' \models \mathsf{Shift}(t, ac) \leftrightarrow m = m' \circ t \models ac$ (see [31]).

$$\begin{tikzcd}
ac \triangleright L \arrow[rr, "t"] \arrow[dr, "m"'] & & L' \triangleleft \mathsf{Shift}(t, ac) \arrow[dl, "m'"] \\
& G &
\end{tikzcd}$$

To improve readability, we assume projection functions *ac*, *lhs* and *rhs*, returning, respectively, the application condition, left-hand side and right-hand side of a rule. Thus, given a rule $p = (L \xleftarrow{l} K \xrightarrow{r} R, ac)$, $ac(p) = ac$, $lhs(p) = L$, and $rhs(p) = R$.

We only consider injective matches, that is, monomorphisms. If the match m is understood, a DPO transformation step $G \xRightarrow{p,m} H$ will be simply written $G \xRightarrow{p} H$. A transformation sequence or trace $\rho = \rho_1 \dots \rho_n : G \Rightarrow^* H$ via rules $p_1, \dots, p_n$ is a sequence of transformation steps $\rho_i = (G_i \xRightarrow{p_i, m_i} H_i)$ such that $G_1 = G$, $H_n = H$, and consecutive steps are composable, that is, $G_{i+1} = H_i$ for all $1 \leq i < n$. Given a sequence $\rho = \rho_1 \dots \rho_n$, we define $\rho(i) = \rho_i$, for $1 \leq i \leq n$. The category of transformation sequences over an adhesive category $\mathcal{C}$, denoted by $\mathbf{Trf}(\mathcal{C})$, has all graphs in $|\mathcal{C}|$ as objects and all transformation sequences as arrows.

Next, we introduce the notion of a rule morphism, relating two graph-transformation rules. Parisi-Presicce proposed in [46] a notion of rule morphism similar to the one below, although we consider rules with application conditions, and require the commuting squares to be pullbacks instead of pushouts.

Definition 2. ***(Rule morphism [13])*** *Given rules $p_i = (L_i \xleftarrow{l_i} K_i \xrightarrow{r_i} R_i,\ ac_i)$, for $i = 0, 1$, a rule morphism $f : p_0 \to p_1$ is a tuple $f = (f_L, f_K, f_R)$ of graph monomorphisms $f_L : L_0 \to L_1$, $f_K : K_0 \to K_1$, and $f_R : R_0 \to R_1$ such that the squares with the span morphisms l_0, l_1, r_0, and r_1 are pullbacks, as in the diagram below, and such that $ac_1 \Rightarrow \mathsf{Shift}(f_L, ac_0)$.*

$$\begin{tikzcd}
p_0 \arrow[d, "f"'] & : & ac_0 & \triangleright & L_0 \arrow[d, "f_L"'] \arrow[dr, phantom, "pb"] & K_0 \arrow[l, "l_0"'] \arrow[r, "r_0"] \arrow[d, "f_K"] \arrow[dr, phantom, "pb"] & R_0 \arrow[d, "f_R"] \\
p_1 & : & ac_1 & \triangleright & L_1 & K_1 \arrow[l, "l_1"] \arrow[r, "r_1"'] & R_1
\end{tikzcd}$$

Asking that the two squares are pullbacks means, precisely, to preserve the "structure" of objects. I.e., we preserve what should be deleted, what should be added, and what must remain invariant. Of course, pushouts also preserve the created and deleted parts, but they reflect this structure as well, which we do not want in general. With componentwise identities and composition, rule morphisms define the category $\mathbf{Rule}$.

A key concept in the constructions in Section 2.5 is that of *rule amalgamation* [1]. The amalgamation of two rules p_1 and p_2 glues them together into a single rule $\tilde{p}$ to obtain the effect of the original rules. I.e., the simultaneous application of p_1 and p_2 yields the same successor graph as the application of the amalgamated rule $\tilde{p}$. The possible overlapping of rules p_1 and p_2 is captured by a rule p_0 and rule morphisms $f : p_0 \to p_1$ and $g : p_0 \to p_2$.

Definition 3. ***(Rule amalgamation [13])*** *Given rules $p_i = (L_i \xleftarrow{l_i} K_i \xrightarrow{r_i} R_i, ac_i)$, for $i = 0, 1, 2$, and rule morphisms $f : p_0 \to p_1$ and $g : p_0 \to p_2$, the amalgamated rule $p_1 +_{p_0} p_2$ is the rule $(L \xleftarrow{l} K \xrightarrow{r} R, ac)$ in the diagram below, where subdiagrams (1), (2) and (3) are pushouts, l and r are induced by the universal property of (2) so that all subdiagrams commute, and $ac = \mathit{Shift}(\widehat{f}_L, ac_2) \wedge \mathit{Shift}(\widehat{g}_L, ac_1)$.*

$$\begin{tikzcd}
 & & ac_0 \triangleright L_0 \arrow[dr, "g_L"] \arrow[ddl, "f_L"'] & & & & K_0 \arrow[llll, "l_0"'] \arrow[rrrr, "r_0"] \arrow[dr] \arrow[ddl] & & & & R_0 \arrow[dr] \arrow[ddl] & \\
 & & & ac_2 \triangleright L_2 \arrow[ddl, "\widehat{f}_L"] & & & & K_2 \arrow[llll, "l_2"'] \arrow[rrrr, "r_2"] \arrow[ddl] & & & & R_2 \arrow[ddl] \\
 & ac_1 \triangleright L_1 \arrow[dr, "\widehat{g}_L"'] & & (1) & & K_1 \arrow[llll, "l_1"] \arrow[rrrr, "r_1"] \arrow[dr] & & (2) & & R_1 \arrow[dr] & (3) & \\
 & & ac \triangleright L & & & & K \arrow[llll, "l"] \arrow[rrrr, "r"'] & & & & R &
\end{tikzcd}$$

Notice that in the above diagram all squares are either pushouts or pullbacks (by the van Kampen property [40]) which means that all their arrows are monomorphisms (by being an adhesive HLR category).

This construction can be extended to amalgamate any number of rules with respect to a given one.

Definition 4. ***(Multiple rule amalgamation)*** *Given graph transformation rules $p_i = (L_i \xleftarrow{l_i} K_i \xrightarrow{r_i} R_i, ac_i)$ and $p'_i = (L'_i \xleftarrow{l'_i} K'_i \xrightarrow{r'_i} R'_i, ac'_i)$, for $0 \leq i \leq n$, and $p = (L \xleftarrow{l} K \xrightarrow{r} R, ac)$, and given rule morphisms $f^i : p_i \to p'_i$ and $g^i : p_i \to p$, for $0 \leq i \leq n$, (see Figure 1) the amalgamated rule $\widehat{p} = p\{\amalg_{p_i} p'_i\}_{0\leq i\leq n}$ is the rule $(\widehat{L} \xleftarrow{\widehat{l}} \widehat{K} \xrightarrow{\widehat{r}} \widehat{R}, \widehat{ac})$, where $\widehat{L}$, $\widehat{K}$ and $\widehat{R}$ are the colimits of $f^i_L : L_i \to L'_i$ and $g^i_L : L_i \to L$, $f^i_K : K_i \to K'_i$ and $g^i_K : K_i \to K$, and $f^i_R : R_i \to R'_i$ and $g^i_R : R_i \to R$, respectively, where $\widehat{l}$ and $\widehat{r}$ are uniquely determined by the universal properties of these colimits, and where $\widehat{ac} = \mathit{Shift}(\widehat{f}_L, ac) \wedge \bigwedge_{0\leq i\leq n} \mathit{Shift}(\widehat{g}^i_L, ac'_i)$.*

Notice that this construction does not coincide with multi-amalgamation, as defined in [26], where n rules are amalgamated given a single kernel rule — the construction in [26] is also provided for HLR categories and for rules with (nested) application conditions. It is not difficult to show, based on the HLR-adhesiveness of our category, that the morphisms $\widehat{g}^i_L : L_i \to L'$, $\widehat{f}_L : L \to L'$, $\widehat{g}^i_K : K_i \to K'$, $\widehat{f}_K : K \to K'$,

$$\begin{tikzcd}
 & & p_n \arrow[rrd, "g^n"] \arrow[dd, "f^n"] & & \\
 & p_1 \arrow[ru, phantom, "\cdots"] \arrow[rrr, "g^1"] \arrow[ddd, "f^1"] & & & p \arrow[ddd, dashed, "\widehat{f}"] \\
p_0 \arrow[rrrru, "g^0"] \arrow[dddd, "f^0"] & & & & \\
 & & p'_n \arrow[rrdd, dashed, "\widehat{g}^n"] & & \\
 & & & & \\
 & p'_1 \arrow[ruu, phantom, "\cdots"] \arrow[rrr, dashed, "\widehat{g}^1"] & & & \widehat{p} \\
p'_0 \arrow[rrrru, dashed, "\widehat{g}^0"] & & & &
\end{tikzcd}$$

Figure 1: Multiple Rule Amalgamation

$\widehat{g}^i_R\colon R_i \to R'$, and $\widehat{f}_R\colon R \to R'$, defined by the corresponding colimits, are monomorphisms. In particular, we have that this multiple amalgamation construction is equivalent to iterating standard amalgamation.

Definition 5. *(**Iterative rule amalgamation**) Given graph transformation rules $p_i = (L_i \xleftarrow{l_i} K_i \xrightarrow{r_i} R_i, ac_i)$ and $p'_i = (L'_i \xleftarrow{l'_i} K'_i \xrightarrow{r'_i} R'_i, ac'_i)$, for $0 \leq i \leq n$, and $p = (L \xleftarrow{l} K \xrightarrow{r} R, ac)$, and given rule morphisms $f^i\colon p_i \to p'_i$ and $g^i\colon p_i \to p$, for $0 \leq i \leq n$, the iterative amalgamated rule $\widehat{p} = p\{\oplus_{p_i} p'_i\}_{0\leq i\leq n}$, together with rule morphisms $\widehat{f}\colon p \to \widehat{p}$ and $\widehat{g}^i\colon p'_i \to \widehat{p}$, for $0 \leq i \leq n$, are inductively defined as follows:*

- *Case $n = 0$: $\widehat{p} = p +_{p_0} p'_0$, $\widehat{f}\colon p \to \widehat{p}$ and $\widehat{g}^0\colon p'_0 \to \widehat{p}$ and $\widehat{ac}$ are, respectively, the rule morphisms and the application condition defined by the amalgamation construction (cf. Definition 3).*

- *Case $n > 0$: If $\widehat{\widehat{p}} = p\{\oplus_{p_i} p'_i\}_{0\leq i\leq n-1}$, with rule morphisms $\widehat{\widehat{f}}\colon p \to \widehat{\widehat{p}}$ and $\widehat{\widehat{g}}^i\colon p'_i \to \widehat{\widehat{p}}$, for $0 \leq i \leq n-1$, then we have rule morphisms $\widehat{\widehat{g}}^n\colon p_n \to \widehat{\widehat{p}}$, with $\widehat{\widehat{g}}^n = \widehat{\widehat{f}} \circ g^n$, and $f^n\colon p_n \to p'_n$. So, we can define $\widehat{p} = \widehat{\widehat{p}} +_{p_n} p'_n$ together with morphisms $\widehat{f}\colon p \to \widehat{p}$ and $\widehat{g}^i\colon p'_i \to \widehat{p}$, for $0 \leq i \leq n$, with $\widehat{f} = h \circ \widehat{\widehat{f}}$ and $\widehat{g}^i = h \circ \widehat{\widehat{g}}^i$, for $0 \leq i \leq n-1$, and where $\widehat{g}^n\colon p'_n \to \widehat{p}$ and $h\colon \widehat{\widehat{p}} \to \widehat{p}$ are the morphisms defined by the amalgamation $\widehat{\widehat{p}} +_{p_n} p'_n$.*

(See Figure 2.)

Proposition 1. *Given transformation rules $p_i = (L_i \overset{l_i}{\leftarrow} K_i \overset{r_i}{\rightarrow} R_i, ac_i)$ and $p'_i = (L'_i \overset{l'_i}{\leftarrow} K'_i \overset{r'_i}{\rightarrow} R'_i, ac'_i)$, for $0 \leq i \leq n$, and $p = (L \overset{l}{\leftarrow} K \overset{r}{\rightarrow} R, ac)$, and given rule morphisms $f^i\colon p_i \to p'_i$ and $g^i\colon p_i \to p$, for $0 \leq i \leq n$,*

$$p\{\oplus_{p_i} p'_i\}_{0\leq i\leq n} = p\{\amalg_{p_i} p'_i\}_{0\leq i\leq n}.$$

PROOF. Direct consequence of the fact that a sequence of pushouts is a colimit. □

$$\begin{tikzcd}
 & & & & p_n \arrow[ddr, "g^n"] \arrow[ddddd, "f^n"] \arrow[ddddddr, dashed, "\widehat{\widehat{g}}^{\,n}"] & \\
 & & & p_{n-1} \arrow[drr, "g^{n-1}"] \arrow[ddddd, "f^{n-1}"] & & \\
 & & \cdots & & & p \arrow[dddd, dashed, "\widehat{\widehat{f}}"] \arrow[dddddddd, dotted, bend left, "\widehat{f}"] \\
 & p_1 \arrow[urrrr, "g^1"] \arrow[ddddd, "f^1"] & & & & \\
p_0 \arrow[uurrrrr, "g^0"] \arrow[ddddd, "f^0"] & & & & & \\
 & & & & p'_n \arrow[dddddr, dotted, "\widehat{g}^{\,n}"] & \\
 & & & p'_{n-1} \arrow[rr, dashed, "\widehat{\widehat{g}}^{\,n-1}"] \arrow[ddddrr, dotted, "\widehat{g}^{\,n-1}"] & & \widehat{\widehat{p}} \arrow[dddd, dotted, "h"] \\
 & & \cdots & & & \\
 & p'_1 \arrow[uurrrr, dashed, "\widehat{\widehat{g}}^{\,1}"] \arrow[ddrrrr, dotted, "\widehat{g}^{\,1}"] & & & & \\
p'_0 \arrow[uuurrrrr, dashed, "\widehat{\widehat{g}}^{\,0}"] \arrow[drrrrr, dotted, "\widehat{g}^{\,0}"] & & & & & \\
 & & & & & \widehat{p}
\end{tikzcd}$$

Figure 2: Iterative Rule Amalgamation

2.3. Typed graph transformation systems

A (directed unlabeled) *graph* $G = (V, E, s, t)$ is given by a set of nodes (or vertices) V, a set of edges E, and source and target functions $s, t : E \to V$. Given graphs $G_i = (V_i, E_i, s_i, t_i)$, with $i = 1, 2$, a graph homomorphism $f : G_1 \to G_2$ is a pair of functions $(f_V : V_1 \to V_2, f_E : E_1 \to E_2)$ such that $f_V \circ s_1 = s_2 \circ f_E$ and $f_V \circ t_1 = t_2 \circ f_E$. With componentwise identities and composition this defines the category **Graph**.

Given a distinguished graph *TG*, called *type graph*, a *TG-typed graph* (G, g_G), or simply *typed graph* if *TG* is known, consists of a graph G and a typing homomorphism $g_G : G \to TG$ associating with each vertex and edge of G its type in *TG*. However, to enhance readability, we will use simply g_G to denote a typed graph (G, g_G), and when the typing morphism g_G can be considered implicit, we will often refer to a typed graph (G, g_G) just as G. A *TG*-typed graph morphism between *TG*-typed graphs $(G_i, g_i : G_i \to TG)$, with $i = 1, 2$, denoted $f : (G_1, g_1) \to (G_2, g_2)$, (or simply $f : g_1 \to g_2$), is a graph morphism $f : G_1 \to G_2$ which preserves types, i.e., $g_2 \circ f = g_1$.

$$\begin{tikzcd}
& G_2 \arrow[d, "g_2"] & \\
G_1 \arrow[ur, "k"] \arrow[dr, "g_1"'] & & \\
& TG \arrow[r, "f"'] & TG'
\end{tikzcd}$$

(a) Forward retyping functor

$$\begin{tikzcd}
& G_2 \arrow[rr] \arrow[dd, "g_2"] & & G'_2 \arrow[dd, "g'_2"] \\
G_1 \arrow[ur, "k"] \arrow[rr] \arrow[dr, "g_1"'] & & G'_1 \arrow[ur, "k'"] \arrow[dr, "g'_1"] & \\
& TG \arrow[rr, "f"'] & & TG'
\end{tikzcd}$$

(b) Backward retyping functor

Figure 3: Forward and backward retyping functors

Graph$_{TG}$ is the category of *TG*-typed graphs and *TG*-typed graph morphisms, which is the comma category **Graph** over *TG*.

If the underlying graph category is adhesive (resp., adhesive HLR, weakly adhesive) then so are the associated typed categories [16], and therefore all definitions in Section 2.2 apply to them. A *TG*-typed graph transformation rule $p = (L \overset{l}{\leftarrow} K \overset{r}{\rightarrow} R, ac)$ is a span of injective *TG*-typed graph morphisms and a (nested) application condition on L. Given *TG*-typed graph transformation rules $p_i = (L_i \overset{l_i}{\leftarrow} K_i \overset{r_i}{\rightarrow} R_i, ac_i)$, with $i = 1, 2$, a typed rule morphism $f\colon p_1 \rightarrow p_2$ is a tuple (f_L, f_K, f_R) of *TG*-typed graph monomorphisms such that the squares with the span monomorphisms l_i and r_i, for $i = 1, 2$, are pullbacks, and such that $ac_2 \Rightarrow \mathsf{Shift}(f_L, ac_1)$. *TG*-typed graph transformation rules and typed rule morphisms define the category **Rule**$_{TG}$, which is the comma category **Rule** over *TG*.

We are interested in amalgamating GTSs over different type graphs. Following [6], we use forward and backward retyping functors to deal with graphs over different type graphs. A graph morphism $f\colon TG \rightarrow TG'$ induces a forward retyping functor $f^{>}$: **Graph**$_{TG}$ $\rightarrow$ **Graph**$_{TG'}$, with $f^{>}(g_1) = f \circ g_1$ and $f^{>}(k\colon g_1 \rightarrow g_2) = k$ by composition, as shown in the diagram in Figure 3(a). Similarly, such a morphism f induces a backward retyping functor $f^{<}\colon$ **Graph**$_{TG'}$ $\rightarrow$ **Graph**$_{TG}$, with $f^{<}(g'_1) = g_1$ and $f^{<}(k'\colon g'_1 \rightarrow g'_2) = k\colon g_1 \rightarrow g_2$ by pullbacks and mediating morphisms as shown in the diagram in Figure 3(b). Since, as said above, we refer to a *TG*-typed graph $G \rightarrow TG$ just by its typed graph G, leaving *TG* implicit, given a morphism $f\colon TG \rightarrow TG'$, we may refer to the *TG*$'$-typed graph by $f^{>}(G)$. Since we can retype graphs and graph morphisms, we can retype rules. Given a rule p over a type graph *TG* and a graph morphism $f\colon TG \rightarrow TG'$, we will write things like $f^{<}(p)$ and $f^{>}(p)$ denoting, respectively, the backward and forward retyping of rule p.

A typed graph transformation system over a type graph *TG*, is a graph transformation system where the given graph transformation rules are defined over the category of *TG*-typed graphs. Since we deal with GTSs over different type graphs, we will make explicit the given type graph. This means that, from now on, a typed GTS is a triple (TG, P, π) where *TG* is a type graph, P is a set of rule names and π is a function mapping each rule name p into a rule $(L \overset{l}{\leftarrow} K \overset{r}{\rightarrow} R, ac)$ typed over *TG*.

2.4. GTS morphisms and their effects on behaviour

The set of transformation rules of a GTS specifies a behaviour in terms of the derivations obtained via such rules. A GTS morphism defines a relation between its source and target GTSs by providing an association between their type graphs and rules.

Definition 6. *(**GTS morphism [13]**) Given GTSs $GTS_i = (TG_i, P_i, \pi_i)$, for $i = 0, 1$, a GTS morphism $f : GTS_0 \to GTS_1$, with $f = (f_{TG}, f_P, f_r)$, is given by a morphism $f_{TG} : TG_0 \to TG_1$, a surjective mapping $f_P : P_1 \to P_0$ between the sets of rule names, and a family of rule morphisms $f_r = \{f^p : f^{>}_{TG}(\pi_0(f_P(p))) \to \pi_1(p)\}_{p \in P_1}$.*

Given a GTS morphism $f : GTS_0 \to GTS_1$, each rule in GTS_1 extends a rule in GTS_0. However if there are internal computation rules in GTS_1 that do not extend any rule in GTS_0, we can always consider that the empty rule τ is included in GTS_0, and assume that those rules extend the empty rule. Notice that to deal with rule morphisms defined on rules over different type graphs we retype the source rules. Typed GTSs and GTS morphisms define the category **GTS**.

Different GTS morphisms may lead to different relationships between the possible derivations in its source and target GTSs. For example, when a GTS is extended with additional rules and alien elements, e.g., to measure or to verify some property, we need to guarantee that such an extension does not change the behaviour of the original GTS. Specifically, we need to guarantee that the behaviour of the resulting system is exactly the same, that is, that any derivation in the target system was also possible in the source one (behaviour reflection), and that any derivation in the source system also happens in the target one (behaviour preservation).

Given a GTS morphism $f : GTS_0 \to GTS_1$, we say that it *reflects* behaviour if for any derivation that may happen in GTS_1 there exists a corresponding derivation in GTS_0.

Definition 7. *(**Behaviour-reflecting GTS morphism [13]**) Given transformation systems $GTS_i = (TG_i, P_i, \pi_i)$, for $i = 0, 1$, a GTS morphism $f : GTS_0 \to GTS_1$ reflects behaviour if for all graphs G, H in $|\mathbf{Graph}_{TG_1}|$, all rules p in P_1, and all matches $m : lhs(\pi_1(p)) \to G$ such that $G \overset{p,m}{\Longrightarrow} H$, then $f^{<}_{TG}(G) \xRightarrow{f_P(p), f^{<}_{TG}(m)} f^{<}_{TG}(H)$ in GTS_0.*

Proposition 2. *The composition of behaviour-reflecting GTS morphisms is behaviour-reflecting.*

PROOF. Follows trivially from the fact that, given GTS morphisms $f : GTS_0 \to GTS_1$ and $g : GTS_1 \to GTS_2$, $g^{<}_{TG} \circ f^{<}_{TG} = (g_{TG} \circ f_{TG})^{<}$. □

A useful type of GTS morphisms are those that only add to the transformation rules elements not in their source type graph.

Definition 8. *(**Extension GTS morphism [13]**) Given GTSs $GTS_i = (TG_i, P_i, \pi_i)$, for $i = 0, 1$, a GTS morphism $f : GTS_0 \to GTS_1$, with $f = (f_{TG}, f_P, f_r)$, is an extension morphism if f_{TG} is a monomorphism and for each $p \in P_1$, $\pi_0(f_P(p)) \equiv f^{<}_{TG}(\pi_1(p))$.*

A morphism may be very easily checked to be an extension by making sure that the features "added" in the rules are removed by the backward retyping functor and that the application conditions are equivalent. This may also be useful given that, as shown in the following lemma, an extension GTS morphism is indeed a behaviour-reflecting GTS morphism.

Lemma 1. *(**From [13]**) All extension GTS morphisms are behaviour-reflecting.*

Behaviour-reflecting morphisms not only reflect individual transformation steps, they reflect entire traces.

Fact 1. *Given a behaviour-reflecting GTS morphism $f : GTS_0 \to GTS_1$ and a trace in GTS_1 $\rho = \rho_1 \dots \rho_n$, with $\rho_i = G_i \overset{p_i, m_i}{\Longrightarrow} G_{i+1}$, for $i = 1 \dots n-1$, there is a corresponding trace $f^{<}_{TG}(\rho)$ in GTS_0, with $f^{<}_{TG}(\rho_i) = f^{<}_{TG}(G_i) \overset{f_P(p_i), f^{<}_{TG}(m_i)}{\Longrightarrow} f^{<}_{TG}(G_{i+1})$, for $i = 1 \dots n-1$.*

We also need to characterize morphisms that preserve behaviour. We find in the literature definitions of behaviour-preserving morphisms as morphisms in which the rules in the source GTS are included in the set of rules of the target GTS (see, e.g., [33, 29]). Although these morphisms trivially preserve behaviour, they are not useful for our purposes. In our case, in addition to adding new rules, we may be enriching the rules themselves. Consider, e.g., a *GTS*$_1$ which extends *GTS*$_0$ just by an additional element type, a step-counter, and extends each rule by adding a step-counter element which gets incremented. Without an additional initialisation rule creating the step-counter set to zero, we must make sure the step-counter is also added to the initial configuration. Otherwise, none of the rules in the extended system would be applicable on a graph without a step-counter object. In other words, we cannot only consider relations between rules, but need to consider relations between traces. We, however, do not need to consider general traces, we may focus on some form of *stuttering simulation* [42, 43], in a very specific form of matching of traces. Intuitively, in a stuttering simulation we do not need traces to match completely, there might be steps that match with identity steps.

Definition 9. *(**Matching traces**) Let $GTS_i = (TG_i, P_i, \pi_i)$, for $i = 0, 1$, be graph transformation systems, and let $f : GTS_0 \to GTS_1$ be a GTS morphism. Given traces $\rho^0 : G^0 \Rightarrow^* H^0$ in GTS_0 and $\rho^1 : G^1 \Rightarrow^* H^1$ in GTS_1, we say that ρ^1 f-*matches ρ^0 if $f^{<}_{TG}(G^1) = G^0$ and there is a strictly increasing function $\alpha : \mathbb{N} \to \mathbb{N}$ such that, for all $i \in \mathbb{N}$, $\rho^0(i) = f^{<}_{TG}(\rho^1(\alpha(i)))$, that is, if $\rho^0(i) = G^0_i \overset{p^0_i, m^0_i}{\Longrightarrow} G^0_{i+1}$ and $\rho^1(\alpha(i)) = G^1_{\alpha(i)} \overset{p^1_{\alpha(i)}, m^1_{\alpha(i)}}{\Longrightarrow} G^1_{\alpha(i)+1}$, then $G^0_i = f^{<}_{TG}(G^1_{\alpha(i)})$, $G^0_{i+1} = f^{<}_{TG}(G^1_{\alpha(i)+1})$, $p^0_i = f_P(p^1_{\alpha(i)})$, and $m^0_i = f^{<}_{TG}(m^1_{\alpha(i)})$.*

For example, the following diagram shows the beginning of two matching traces, where related transformation steps are joined by dashed lines, and α is defined as $\alpha(0) = 0$, $\alpha(1) = 2$, $\alpha(2) = 3, \dots$

$$\begin{tikzcd}
\rho^0 & G^0_0 \arrow[r, "\rho^0_0", ""{name=A0, below}] & G^0_1 \arrow[r, "\rho^0_1", ""{name=A1, below}] & G^0_2 \arrow[r, "\rho^0_2", ""{name=A2, below}] & G^0_3 \arrow[r, "\rho^0_3"] & G^0_4 \arrow[r] & \dots \\
\rho^1 & G^1_0 \arrow[r, "\rho^1_0"', ""{name=B0}] & G^1_1 \arrow[r, "\rho^1_1"'] & G^1_2 \arrow[r, "\rho^1_2"', ""{name=B2}] & G^1_3 \arrow[r, "\rho^1_3"', ""{name=B3}] & G^1_4 \arrow[r] & \dots
\arrow[from=A0, to=B0, dash, dashed]
\arrow[from=A1, to=B2, dash, dashed]
\arrow[from=A2, to=B3, dash, dashed]
\end{tikzcd}$$

The intuition behind matching traces is that, if we take any trace ρ in *GTS*$_1$ and remove the extra elements introduced by the GTS morphism from the consecutive graphs in

ρ, we get a trace in GTS_0, with the additional steps in GTS_1 going to identity steps in GTS_0. Note that if a trace ρ^1 f-matches ρ^0, the 'extra' steps in ρ^1 may only make changes on elements not in TG_0, that is, elements introduced in the extension. For instance, the above derivation ρ^1_1 induces an identity step on GTS_0, that is, $f^{<}_{TG}(\rho^1_1) = G^0_1 \overset{\tau}{\Longrightarrow} G^0_1$.

Definition 10. ***(Extended trace)*** *Given transformation systems $GTS_i = (TG_i, P_i, \pi_i)$, for $i = 0, 1$, a GTS morphism $f : GTS_0 \to GTS_1$, and traces ρ^0 in GTS_0 and ρ^1 in GTS_1 such that ρ^1 f-matches ρ^0, with some strictly increasing function α, the* extended trace *of ρ^0, denoted $\overline{\rho^0}$, is ρ^0 with identity steps (applications of τ rules) in those positions in which α^{-1} is not defined.*

For f-matching traces ρ^0 and ρ^1, the possibility of extending traces leaves us with a one-to-one correspondence between the traces.

$$\begin{tikzcd}
\rho^0 & G^0_0 \arrow[r] & G^0_1 \arrow[rr] & & G^0_2 \arrow[r] & G^0_3 \arrow[r] & G^0_4 \arrow[r] & \ldots \\
\overline{\rho^0} & G^0_0 \arrow[r, "\rho^0_0"] \arrow[d, dashed, no head, shift left=6ex] & G^0_1 \arrow[r, "\tau"] \arrow[d, dashed, no head, shift left=6ex] & G^0_1 \arrow[r, "\rho^0_1"] \arrow[d, dashed, no head, shift left=6ex] & G^0_2 \arrow[r, "\rho^0_2"] \arrow[d, dashed, no head, shift left=6ex] & G^0_3 \arrow[r, "\rho^0_3"] \arrow[d, dashed, no head, shift left=6ex] & G^0_4 \arrow[r] & \ldots \\
\rho^1 & G^1_0 \arrow[r] & G^1_1 \arrow[r] & G^1_2 \arrow[r] & G^1_3 \arrow[r] & G^1_4 \arrow[r] & G^1_5 \arrow[r] & \ldots
\end{tikzcd}$$

Fact 2. *Given transformation systems $GTS_i = (TG_i, P_i, \pi_i)$, for $i = 0, 1$, a GTS morphism $f : GTS_0 \to GTS_1$, and traces ρ^0 in GTS_0 and ρ^1 in GTS_1 such that ρ^1 f-matches ρ^0, with some strictly increasing function α, then ρ^1 f-matches $\overline{\rho^0}$, with the strictly increasing function $\overline{\alpha}(x) = x$.*

With the formulation in Definition 11, we make sure that the derivations in the source GTS, GTS_0, are applicable on graphs in $|\mathbf{Graph}_{TG_1}|$. The possibility of the extra steps in the target GTS will allow the extended rules to be fired after appropriate changes are made so that their requirements are satisfied.

Definition 11. ***(Behaviour-preserving GTS morphism)*** *Given typed graph transformation systems $GTS_i = (TG_i, P_i, \pi_i)$, for $i = 0, 1$, a GTS morphism $f : GTS_0 \to GTS_1$ is behaviour-preserving if for each trace ρ^0 in GTS_0 there is a trace ρ^1 in GTS_1 that f-matches ρ^0.*

Definition 9 characterizes f-matching in terms of infinite traces. But in practice, we will find many situations in which we have to deal with finite traces. Theoretically, we may handle finite traces by assuming identity derivations at deadlock states, using the empty rule, so that we do not have blocking states. Given a GTS morphism $f : GTS_0 \to GTS_1$, and traces ρ^0 in GTS_0 and ρ^1 in GTS_1 such that ρ^1 f-matches ρ^0, with some strictly increasing function α, if there is a deadlock state in a trace ρ^0 in GTS_0 at some position i, with $\alpha(i) = i + k$ for some k greater or equal than 0, we would expect that if f preserves behaviour, there was a corresponding sequence of identity derivations in GTS_1 with $\alpha(j) = j + k$, for all j greater or equal than i. However, this may be not always the case, since we may have non-terminating computations in GTS_1 due to newly introduced rules, mapping to an infinite sequence of identity derivations, thus having the same situation for a deadlock and a livelock.

Definition 12. *(**Deadlock-preserving GTS morphism**) Given transformation systems $GTS_i = (TG_i, P_i, \pi_i)$, for $i = 0, 1$, a GTS morphism $f\colon GTS_0 \to GTS_1$ is deadlock-preserving if for each trace ρ^0 in GTS_0 that leads to a deadlock there is no infinite trace ρ^1 in GTS_1 that f-matches ρ^0.*

In order to check that a morphism is deadlock preserving, it is sufficient to prove that the set of transformation rules p that map to the empty rule, that is, $f^{<}_{TG}(\pi_1(p)) = \tau$, is terminating.

Definition 13. *(**Strong behaviour-preserving GTS morphism**) Given transformation systems $GTS_i = (TG_i, P_i, \pi_i)$, for $i = 0, 1$, a GTS morphism $f\colon GTS_0 \to GTS_1$ strongly preserves behaviour if it preserves both behaviour and deadlocks.*

Strong behaviour-preserving morphisms compose as expected.

Proposition 3. *The composition of strong behaviour-preserving GTS morphisms is strong behaviour-preserving.*

PROOF. Compositionality of behaviour preservation trivially follows from the facts that, given GTS morphisms $f\colon GTS_0 \to GTS_1$ and $g\colon GTS_1 \to GTS_2$, $g^{<}_{TG} \circ f^{<}_{TG} = (g_{TG} \circ f_{TG})^{<}$, and the composition of strictly increasing functions is strictly increasing. Now, suppose a trace ρ^0 in GTS_0 that leads to a deadlock. Since f preserves deadlocks, all traces ρ^1 in GTS_1 that f-match ρ^0 are finite. And similarly, because g also preserves deadlocks, there cannot be any infinite trace in GTS_2 that g-matches any of these ρ^1. $\square$

The definition of behaviour-protecting GTS morphism establishes a bidirectional correspondence between derivations in the source and target GTSs.

Definition 14. *(**Behaviour-protecting GTS morphism**) Given transformation systems $GTS_i = (TG_i, P_i, \pi_i)$, for $i = 0, 1$, a GTS morphism $f\colon GTS_0 \to GTS_1$ is behaviour-protecting if it reflects and strongly preserves behaviour.*

This definition of behaviour-protecting morphisms is different to the one in [13]. According to the definition in [13], a GTS morphism $f : GTS_0 \to GTS_1$ is behaviour-protecting if for all graphs G and H in $|\mathbf{Graph}_{TG_1}|$, all rules p in P_1, and all matches $m : lhs(\pi_1(p)) \to G$, we have $g^{<}_{TG}(G) \xRightarrow{g_P(p), g^{<}_{TG}(m)} g^{<}_{TG}(H) \iff G \overset{p,m}{\Longrightarrow} H$. This definition is too strict, not considering as protecting some morphisms that intuitively are, as for example an extension in which additional elements in the graphs are introduced by additional rules. With the formulation in [13], we may have cases in which the morphisms are protecting in a rule basis, but not when looking at traces, which may be blocked. Consider again the above step-counter example, with the formulation in Definition 11, we make sure the derivations in the source GTS GTS_0 are applicable on graphs in $|\mathbf{Graph}_{TG_0}|$.

Proposition 4. *The composition of behaviour-protecting GTS morphisms is behaviour-protecting.*

PROOF. Follows from Propositions 2 and 3. □

Behaviour-protecting GTS morphisms reflect and preserve behaviour, including deadlocks and livelocks.

Fact 3. *Given a behaviour-protecting GTS morphism, there is a deadlock (resp., livelock) in the target GTS if and only if there is a corresponding deadlock (resp., livelock) in the source GTS.*

PROOF. Assume a behaviour-protecting GTS morphism $f\colon \mathit{GTS}_0 \to \mathit{GTS}_1$. Since f protects behaviour, it preserves deadlocks. Now, assume a graph G in $|\mathbf{Graph}_{TG_1}|$, with no rule in P_1 applicable on G (other than the empty rule). If there were a rule p in P_0 applicable on $f^{<}_{TG}(G)$, then, since f preserves behaviour, there would be a corresponding trace in GTS_1.

Since f preserves behaviour, any infinite trace ρ^0 in GTS_0 has a corresponding infinite trace ρ^1 in GTS_1. We are then left with reflection of livelocks. Assume an infinite trace ρ^1 in GTS_1 such that $f^{<}_{TG}(\rho_1)$ is finite. The only possibility is that an infinite number of consecutive derivations in the trace $\rho^1_k \Rightarrow \rho^1_{k+1} \Rightarrow \ldots$, starting for some k on some graph G, maps to an infinite sequence of identity derivations in GTS_0 from graph $f^{<}_{TG}(G)$, that is, all derivations $f^{<}_{TG}(\rho^1_j)$, for $j \geq k$, correspond to applications of the empty rule. But this contradicts the fact that f is deadlock preserving: there cannot be a finite trace in GTS_0 that f-matches an infinite one in GTS_1. □

2.5. GTS amalgamations

GTS amalgamation provides a very convenient way of composing GTSs. By amalgamating GTSs with respect to a kernel GTS, we are not only putting type graphs and rules together, we are amalgamating rules so that we can synchronise their behaviours in a very precise way. The definition below shows how we can compose two GTSs that share some common part, and Theorem 1 establishes behaviour-related properties on the induced morphisms.

Definition 15. ***(GTS Amalgamation [13]).*** *Given typed graph transformation systems $\mathit{GTS}_i = (\mathit{TG}_i, P_i, \pi_i)$, for $i = 0, 1, 2$, and GTS morphisms $f\colon \mathit{GTS}_0 \to \mathit{GTS}_1$ and $g\colon \mathit{GTS}_0 \to \mathit{GTS}_2$, the amalgamated GTS $\widehat{\mathit{GTS}} = \mathit{GTS}_1 +_{\mathit{GTS}_0} \mathit{GTS}_2$ is the GTS $(\widehat{\mathit{TG}}, \widehat{P}, \widehat{\pi})$ constructed as follows. We first construct the pushout of typing graph morphisms $f_{TG}\colon \mathit{TG}_0 \to \mathit{TG}_1$ and $g_{TG}\colon \mathit{TG}_0 \to \mathit{TG}_2$, obtaining morphisms $\widehat{f}_{TG}\colon \mathit{TG}_2 \to \widehat{\mathit{TG}}$ and $\widehat{g}_{TG}\colon \mathit{TG}_1 \to \widehat{\mathit{TG}}$. The pullback of set morphisms $f_P\colon P_1 \to P_0$ and $g_P\colon P_2 \to P_0$ defines morphisms $\widehat{f}_P\colon \widehat{P} \to P_2$ and $\widehat{g}_P\colon \widehat{P} \to P_1$. Then, for each rule p in $\widehat{P}$, the rule $\widehat{\pi}(p)$ is defined as the amalgamation of rules $\widehat{f}^{>}_{TG}(\pi_2(\widehat{f}_P(p)))$ and $\widehat{g}^{>}_{TG}(\pi_1(\widehat{g}_P(p)))$ with respect to the kernel rule $\widehat{f}^{>}_{TG}(g^{>}_{TG}(\pi_0(g_P(\widehat{f}_P(p)))))$.*

$$\begin{tikzcd}
\mathit{GTS}_0 \arrow[r, "f"] \arrow[d, "g"'] & \mathit{GTS}_1 \arrow[d, dotted, "\widehat{g}"] \\
\mathit{GTS}_2 \arrow[r, dotted, "\widehat{f}"] & \widehat{\mathit{GTS}}
\end{tikzcd}$$

Since the amalgamation of GTSs is the basic construction for combining them, it is very important to know whether the reflection of derivations remains invariant under amalgamations.

Proposition 5. *(**From [13]**) Given transformation systems $GTS_i = (TG_i, P_i, \pi_i)$, for $i = 0, 1, 2$, and the amalgamation $\widehat{GTS} = GTS_1 +_{GTS_0} GTS_2$ of GTS morphisms $f : GTS_0 \to GTS_1$ and $g : GTS_0 \to GTS_2$ (see diagram in Definition 15), if f_{TG} is a monomorphism and g is an extension morphism, then $\widehat{g}$ is also an extension morphism.*

Proposition 6. *Given transformation systems $GTS_i = (TG_i, P_i, \pi_i)$, for $i = 0, 1, 2$, and the amalgamation $\widehat{GTS} = GTS_1 +_{GTS_0} GTS_2$ of GTS morphisms $f : GTS_0 \to GTS_1$ and $g : GTS_0 \to GTS_2$ (see diagram in Definition 15), if f is a behaviour-reflecting GTS morphism, f_{TG} is a monomorphism, and g is a strongly behaviour-preserving morphism, then $\widehat{g}$ is also a strongly behaviour-preserving morphism.*

PROOF. Let ρ^1 be a trace in GTS_1. Since f is a behaviour-reflecting GTS morphism, there is a corresponding trace $\rho^0 = f^{<}_{TG}(\rho^1)$ in GTS_0. Since g preserves behaviour, there is a trace ρ^2 in GTS_2 that f-matches ρ^0, with some strictly increasing function α. Let $\overline{\rho^0}$ be the extension of trace ρ^0 (see Definition 10), and let us extend ρ^1 accordingly to get $\overline{\rho^1}$, by introducing identity steps in those positions in which α^{-1} is not defined.

Thus, we have that for derivations $G^1_i \xRightarrow{p^1_i, m^1_i} G^1_{i+1}$ in $\overline{\rho^1}$ and $G^2_i \xRightarrow{p^2_i, m^2_i} G^2_{i+1}$ in $\overline{\rho^2}$, for $i = 1 \dots n$, there is a corresponding derivation in GTS_0, $G^0_i \xRightarrow{p^0_i, m^0_i} G^0_{i+1}$, where the rule $p^0_i = f_P(p^1_i) = g_P(p^2_i)$ can be applied on $G^0_i = f^{<}_{TG}(G^1_i) = g^{<}_{TG}(G^2_i)$ with match $m^0_i = f^{<}_{TG}(m^1_i) = g^{<}_{TG}(m^2_i)$ satisfying the application condition of rule $\pi_0(p^0_i) = \pi_0(f_P(p^1_i)) = \pi_0(g_P(p^2_i))$, and resulting in a graph $G^0_{i+1} = f^{<}_{TG}(G^1_{i+1}) = g^{<}_{TG}(G^2_{i+1})$.

Note that by the amalgamation construction in Definition 15, the set of rules of $\widehat{GTS}$ includes, for each p in $\widehat{P}$, the amalgamation of (the forward retyping of) the rules $\pi_1(\widehat{g}_P(p)) = (L_1 \xleftarrow{l_1} K_1 \xrightarrow{r_1} R_1, ac_1)$ and $\pi_2(\widehat{f}_P(p)) = (L_2 \xleftarrow{l_2} K_2 \xrightarrow{r_2} R_2, ac_2)$, with kernel rule $\pi_0(f_P(\widehat{g}_P(p))) = \pi_0(g_P(\widehat{f}_P(p))) = (L_0 \xleftarrow{l_0} K_0 \xrightarrow{r_0} R_0, ac_0)$.

For any $\widehat{TG}$ graph G, G is the pushout of the graphs $\widehat{g}^{<}_{TG}(G)$, $\widehat{f}^{<}_{TG}(G)$ and $f^{<}_{TG}(\widehat{g}^{<}_{TG}(G))$ (with the obvious morphisms). This can be proved using a van Kampen square, where in the bottom we have the pushout of the type graphs, the vertical faces are the pullbacks defining the backward retyping functors and on top we have that pushout.

Thus, for each graph G_i in $\widehat{GTS}$, if a transformation rule in GTS_1 can be applied on $\widehat{g}^{<}_{TG}(G_i)$, the corresponding transformation rule should be applicable on G_i in $\widehat{GTS}$. The diagram in Figure 4 focus on the lefthand sides of the involved rules.

As we have seen above, rules $\widehat{g}_P(p_i)$, $\widehat{f}_P(p_i)$, and $\widehat{f}_P(g_P(p_i)) = \widehat{g}_P(f_P(p_i))$ are applicable on their respective graphs using the matchings depicted in the above diagram. Since, by the amalgamation construction, the top square is a pushout, and $g_1 \circ \widehat{g}^{>}_{TG}(m^1_i) \circ \widehat{g}^{>}_{TG}(f^{\widehat{g}_P(p_i)}_L) = g_2 \circ \widehat{f}^{>}_{TG}(m^2_i) \circ \widehat{f}^{>}_{TG}(g^{\widehat{f}_P(p_i)}_L)$, then there is a unique morphism $m_i : L_i \to G_i$ making $g_1 \circ \widehat{g}^{>}_{TG}(m^1_i) = m_i \circ \widehat{g}^{p_i}_L$ and $g_2 \circ \widehat{f}^{>}_{TG}(m^2_i) = m_i \circ \widehat{f}^{p_i}_L$. This m_i will be used as matching morphism in the derivation we seek.

$$\begin{tikzcd}
 & & \widehat{f}^{>}_{TG}(g^{>}_{TG}(L_i^0)) = \widehat{g}^{>}_{TG}(f^{>}_{TG}(L_i^0)) \arrow[dll, "\widehat{f}^{>}_{TG}(g_L^{\widehat{f}_P(p_i)})"'] \arrow[d, "\widehat{f}^{>}_{TG}(g^{>}_{TG}(m_i^0)) = \widehat{g}^{>}_{TG}(f^{>}_{TG}(m_i^0))"] \arrow[drr, "\widehat{g}^{>}_{TG}(f_L^{\widehat{g}_P(p_i)})"] & & \\
\widehat{f}^{>}_{TG}(L_i^2) \arrow[d, "f^{>}_{TG}(m_i^2)"] \arrow[drr, "\widehat{f}_L^{p_i}"'] & & \widehat{f}^{>}_{TG}(g^{>}_{TG}(g^{<}_{TG}(\widehat{f}^{<}_{TG}(G_i^0)))) = \widehat{g}^{>}_{TG}(f^{>}_{TG}(f^{<}_{TG}(\widehat{g}^{<}_{TG}(G_i^0)))) \arrow[dll] \arrow[drr] & & \widehat{g}^{>}_{TG}(L_i^1) \arrow[d, "\widehat{g}^{>}_{TG}(m_i^1)"] \arrow[dll, "\widehat{g}_L^{p_i}"] \\
\widehat{f}^{>}_{TG}(\widehat{f}^{<}_{TG}(G_i)) \arrow[drr, "g_2"'] & & L_i \arrow[d, "m_i"] & & \widehat{g}^{>}_{TG}(\widehat{g}^{<}_{TG}(G_i)) \arrow[dll, "g_1"] \\
 & & G_i & &
\end{tikzcd}$$

Figure 4: Applicability of the transformation rule in $\widehat{GTS}$

By construction, the application condition ac_i of the amalgamated rule p_i is the conjunction of the shiftings of the application conditions of $\widehat{g}_P(p_i)$ and $\widehat{f}_P(p_i)$. Then, since

$$m_i^1 \models ac_i^1 \iff m_i \models \mathsf{Shift}(\widehat{g}_L^{p_i}, ac_i^1)$$

and

$$m_i^2 \models ac_i^2 \iff m_i \models \mathsf{Shift}(\widehat{f}_L^{p_i}, ac_i^2),$$

and therefore

$$m_i^1 \models ac_i^1 \wedge m_i^2 \models ac_i^2 \iff m_i \models ac_i.$$

We can then conclude that rule p_i is applicable on graph G_i with match m_i satisfying its application condition ac_i. Indeed, given the rule $\pi(p_i) = (L_i \xleftarrow{l_i} K_i \xrightarrow{r_i} R_i, ac_i)$ we have the following derivation:

$$\begin{tikzcd}
ac_i \arrow[r, phantom, "\triangleright"] & L_i \arrow[d, "m_i"'] \arrow[dr, phantom, "po"] & K_i \arrow[l, "l_i"'] \arrow[r, "r_i"] \arrow[d] \arrow[dr, phantom, "po"] & R_i \arrow[d] \\
 & G_i & D_i \arrow[l] \arrow[r] & G_{i+1}
\end{tikzcd}$$

Let us finally check then that D_i and G_{i+1} are as expected. To improve readability, in the following diagrams we eliminate the forward retyping functors. For instance, for the rest of the theorem L_i^0 denotes $\widehat{f}^{>}_{TG}(g^{>}_{TG}(L_i^0)) = \widehat{g}^{>}_{TG}(f^{>}_{TG}(L_i^0))$, L_i^1 denotes $\widehat{g}^{>}_{TG}(L_i^1)$, etc.

First, let us focus on the pushout complement of $l_i : K_i \to L_i$ and $m_i : L_i \to G_i$. Given rules $\widehat{g}_P(p_i)$, $\widehat{f}_P(p_i)$, and $\widehat{f}_P(g_P(p_i)) = \widehat{g}_P(f_P(p_i))$ and rule morphisms between them as above, the diagram in Figure 5 shows both the construction by amalgamation of the morphism $l_i : K_i \to L_i$, and the construction of the pushout complements for morphisms l_i^j and m_i^j, for $i = 0, 1, 2$.

$$\begin{tikzcd}
 & & L_i^0 \arrow[dll] \arrow[ddr] \arrow[dddd, "m_i^0"'] & & & & K_i^0 \arrow[llll, "l_i^0"'] \arrow[dll] \arrow[ddr] \arrow[dddd, dashed] & \\
L_i^2 \arrow[ddr] \arrow[dddd, "m_i^2"'] & & & & K_i^2 \arrow[llll, "l_i^2"] \arrow[ddr] \arrow[dddd, dashed] & & & \\
 & & & L_i^1 \arrow[dll] \arrow[dddd, "m_i^1"'] & & & & K_i^1 \arrow[llll, "l_i^1"'] \arrow[dll] \arrow[dddd, dashed] \\
 & L_i \arrow[dddd, "m_i"] \arrow[dddddl, dotted] & & & & K_i \arrow[llll, "l_i"'] \arrow[dddd, dashed] & & \\
 & & G_i^0 \arrow[dll] \arrow[ddr] & & & & D_i^0 \arrow[llll, dashed, "\widehat{l_i^0}"'] \arrow[dll, dashed] \arrow[ddr, dashed] & \\
G_i^2 \arrow[ddr] & & & & D_i^2 \arrow[llll, dashed, "\widehat{l_i^2}"'] \arrow[ddr, dashed] & & & \\
 & & & G_i^1 \arrow[dll] & & & & D_i^1 \arrow[llll, dashed, "\widehat{l_i^1}"'] \arrow[dll, dashed] \\
 & G_i & & & & D_i \arrow[llll, dashed, "\widehat{\widehat{l}}"'] \arrow[dlllll, dotted] & & \\
X & & & & & & &
\end{tikzcd}$$

Figure 5: Construction of the morphisms l_i, l_i^j and m_i^j, for $i = 0, 1, 2$

By the pushout of $D_i^0 \to D_i^1$ and $D_i^0 \to D_i^2$, and given the commuting subdiagram

$$\begin{tikzcd}
 & & & & & D_i^0 \arrow[dll] \arrow[ddr] & \\
G_i^2 \arrow[ddr] & & & D_i^2 \arrow[lll] \arrow[ddrr] & & & \\
 & & G_i^1 \arrow[dl] & & & & D_i^1 \arrow[llll] \arrow[dl] \\
 & G_i & & & & D_i \arrow[llll, dashed] &
\end{tikzcd}$$

there exists a unique morphism $D_i \to G_i$ making the diagram commute. This D_i is indeed the object of the pushout complement we were looking for. By the pushout of $K_i^0 \to K_i^1$ and $K_i^0 \to K_i^2$, there is a unique morphism from K_i to D_i making the diagram commute. We claim that these morphisms $K_i \to D_i$ and $D_i \to G_i$ are the pushout complement of $K_i \to L_i$ and $L_i \to G_i$. Suppose that the pushout of $K_i \to L_i$ and $K_i \to D_i$ were $L_i \to X$ and $D_i \to X$ for some graph X different from G_i. By the pushout of $K_i^1 \to D_i^1$ and $K_i^1 \to L_i^1$ there is a unique morphism $G_i^1 \to X$ making the diagram commute. By the pushout of $K_i^2 \to D_i^2$ and $K_i^2 \to L_i^2$ there is a unique morphism $G_i^2 \to X$ making the diagram commute. By the pushout of $G_i^0 \to G_i^1$ and $G_i^0 \to G_i^2$, there is a unique morphism $G_i \to X$. But since $L_i \to X$ and $D_i \to X$ are the pushout of $K_i \to L_i$ and $K_i \to D_i$, there is a unique morphism $X \to G_i$ making the diagram commute. Therefore, we can conclude that X and G_i are isomorphic.

By a similar construction for the righthand sides we get the pushout

$$\begin{tikzcd} K_i \arrow[r] \arrow[d] \arrow[dr, phantom, "po"] & R_i \arrow[d] \\ D_i \arrow[r] & G_{i+1} \end{tikzcd}$$

and therefore the derivation $G_i \xRightarrow{p_i, m_i} G_{i+1}$.

The trace ρ thus obtained $\widehat{g}$-matches the trace $\overline{\rho^1}$ with strictly increasing function $\beta(x) = x$. ρ also $\widehat{g}$-matches the trace ρ^1 with $\beta = \alpha$, since the positions of identity derivations introduced in ρ^1 to get $\overline{\rho^1}$ are the same as to get $\overline{\rho^0}$ from ρ^0. We can therefore conclude that $\widehat{g}$ is a behaviour-preserving morphism.

Let us finally see that $\widehat{g}$ also preserves deadlocks. Above, we have seen that any trace ρ in $\widehat{GTS}$ comes from traces $\rho^1 = \widehat{g}^{<}_{TG}(\rho)$ in GTS_1, $\rho^2 = \widehat{f}^{<}_{TG}(\rho)$ in GTS_2, and $\rho^0 = f^{<}_{TG}(\rho^1) = g^{<}_{TG}(\rho^2)$ in GTS_0. Indeed, we have seen that ρ $\widehat{g}$-matches ρ^1, for some strictly increasing function α, and that ρ^2 g-matches ρ^0, with the same α. Let us assume that ρ^1 is finite, that is, there is some $k \geq 0$ such that the sub-trace $\rho^1_k \Rightarrow \rho^1_{k+1} \Rightarrow \ldots$ is a sequence of identity derivations. ρ^0 is also finite, since the above sub-trace leads to the sub-trace of identity derivations $f^{<}_{TG}(\rho^1_k) \Rightarrow f^{<}_{TG}(\rho^1_{k+1}) \Rightarrow \ldots$ in ρ^0. Notice, however, that if a derivation ρ_i is not an identity step and ρ^1_i is an identity step, ρ^2_i cannot be an identity step. And therefore, since ρ is infinite and ρ^1 is finite we have that ρ^2 is an infinite derivation. Since g preserves deadlocks, we cannot have a finite trace ρ^0 that g-matches an infinite trace ρ^1. □

The following result gives conditions under which the morphisms induced by the amalgamation construction can be expected to be protecting.

Theorem 1. *Given typed transformation systems $GTS_i = (TG_i, P_i, \pi_i)$, for $i = 0, 1, 2$, and the amalgamation $\widehat{GTS} = GTS_1 +_{GTS_0} GTS_2$ of GTS morphisms $f\colon GTS_0 \to GTS_1$ and $g\colon GTS_0 \to GTS_2$, if f is a behaviour-reflecting GTS morphism, f_{TG} is a monomorphism, and g is a behaviour-protecting extension morphism, then $\widehat{g}$ is also a behaviour-protecting extension.*

$$\begin{tikzcd} GTS_0 \arrow[r, "f"] \arrow[d, "g"'] & GTS_1 \arrow[d, dotted, "\widehat{g}"] \\ GTS_2 \arrow[r, dotted, "\widehat{f}"] & \widehat{GTS} \end{tikzcd}$$

PROOF. Follows from Propositions 5 and 6. □

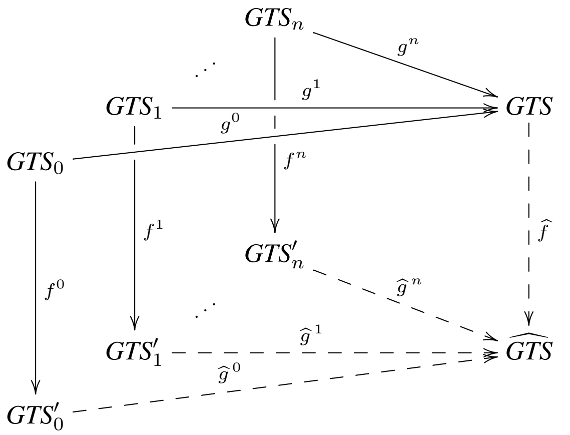
2.6. Multiple GTS amalgamations

To amalgamate several GTSs, as in the case of rules, we can extend GTS amalgamation to multiple amalgamation and to iterated amalgamation, which again coincide.

Definition 16. *(**Multiple GTS Amalgamation**). Given typed graph transformation systems $GTS_i = (TG_i, P_i, \pi_i)$ and $GTS'_i = (TG'_i, P'_i, \pi'_i)$, for $0 \leq i \leq n$, and $GTS = (TG, P, \pi)$, and given GTS morphisms $f^i\colon GTS_i \to GTS'_i$ and $g^i\colon GTS_i \to GTS$, the amalgamated GTS $\widehat{GTS} = GTS\{\amalg_{GTS_i} GTS'_i\}_{0 \leq i \leq n}$ is the GTS $(\widehat{TG}, \widehat{P}, \widehat{\pi})$ constructed as follows:*

- *We first construct the colimit of typing graph morphisms $f^i_{TG}\colon TG_i \to TG'_i$ and $g^i_{TG}\colon TG_i \to TG$ obtaining morphisms $\widehat{f}_{TG}\colon TG \to \widehat{TG}$ and $\widehat{g}^i_{TG}\colon TG'_i \to \widehat{TG}$.*
- *The limit of set morphisms $f^i_P\colon P'_i \to P_i$ and $g^i_P\colon P \to P_i$ defines morphisms $\widehat{f}_P\colon \widehat{P} \to P$ and $\widehat{g}^i_P\colon \widehat{P} \to P'_i$.*
- *For each rule p in $\widehat{P}$, the rule $\widehat{\pi}(p)$ is the amalgamation of rules*

$$\widehat{f}^{>}_{TG}(\pi(\widehat{f}_P(p)))\Big\{ \amalg_{\widehat{f}^{>}_{TG}(g^{i>}_{TG}(\pi_i(g^i_P(\widehat{f}_P(p)))))} \widehat{g}^{i>}_{TG}(\pi'_i(\widehat{g}^i_P(p)))\Big\}_{0\le i\le n}$$



$$\begin{tikzcd}
 & & GTS_n \arrow[drr, "g^n"] \arrow[ddd, "f^n"] & & \\
 & GTS_1 \arrow[rrr, "g^1"] \arrow[ddd, "f^1"] & & & GTS \arrow[ddd, dashed, "\widehat{f}"] \\
GTS_0 \arrow[urrrr, "g^0"] \arrow[ddd, "f^0"] & & & & \\
 & & GTS'_n \arrow[drr, dashed, "\widehat{g}^n"] & & \\
 & GTS'_1 \arrow[rrr, dashed, "\widehat{g}^1"] & & & \widehat{GTS} \\
GTS'_0 \arrow[urrrr, dashed, "\widehat{g}^0"] & & & &
\end{tikzcd}$$

Definition 17. ***(Iterative GTS Amalgamation)****. Given typed graph transformation systems $GTS_i = (TG_i, P_i, \pi_i)$ and $GTS'_i = (TG'_i, P'_i, \pi'_i)$, for $0 \le i \le n$, and $GTS = (TG, P, \pi)$, and given GTS morphisms $f^i\colon GTS_i \to GTS'_i$ and $g^i\colon GTS_i \to GTS$, for $0 \le i \le n$, the iterative amalgamation GTS $\widehat{GTS} = GTS\{\oplus_{GTS_i} GTS'_i\}_{0\le i\le n}$, together with GTS morphisms $\widehat{f}\colon GTS \to \widehat{GTS}$ and $\widehat{g}^i\colon GTS'_i \to \widehat{GTS}$, are defined inductively as follows:*

- *Case $n = 0$: $\widehat{GTS} = GTS +_{GTS_0} GTS'_0$, and $\widehat{f}\colon GTS \to \widehat{GTS}$ and $\widehat{g}^0\colon GTS'_0 \to \widehat{GTS}$ are the GTS morphisms defined by the amalgamation construction (cf. Definition 15).*
- *Case $n > 0$: Let $\widehat{\widehat{GTS}}$ be the amalgamated GTS $GTS\{\oplus_{GTS_i} GTS'_i\}_{0\le i\le n-1}$, with morphisms $\widehat{\widehat{f}}\colon GTS \to \widehat{\widehat{GTS}}$ and $\widehat{\widehat{g}}^i\colon GTS'_i \to \widehat{\widehat{GTS}}$, for $0 \le i \le n-1$, then we have morphisms $\widehat{\widehat{g}}^n\colon GTS_n \to \widehat{\widehat{GTS}}$, with $\widehat{\widehat{g}}^n = \widehat{\widehat{f}} \circ g^n$, and $f^n\colon GTS_n \to GTS'_n$. Thus, we define $\widehat{GTS} = \widehat{\widehat{GTS}} +_{GTS_n} GTS'_n$, together with morphisms $\widehat{f}\colon GTS \to \widehat{GTS}$ and $\widehat{g}^i\colon GTS'_i \to \widehat{GTS}$, for $0 \le i \le n$, with $\widehat{f} = h \circ \widehat{\widehat{f}}$ and $\widehat{g}^i = h \circ \widehat{\widehat{g}}^i$, for $0 \le i \le n-1$, and where $\widehat{g}^n\colon GTS'_n \to \widehat{GTS}$ and $h\colon \widehat{\widehat{GTS}} \to \widehat{GTS}$ are the morphisms defined by the amalgamation $\widehat{\widehat{GTS}} +_{GTS_n} GTS'_n$.*

Proposition 7. *Given graph transformation systems GTS_i and GTS'_i, for $0 \le i \le n$, and GTS, and given GTS morphisms $f^i\colon GTS_i \to GTS'_i$ and $g^i\colon GTS_i \to GTS$, for $0 \le i \le n$, $GTS\{\oplus_{GTS_i} GTS'_i\}_{0\le i\le n} = GTS\{\amalg_{GTS_i} GTS'_i\}_{0\le i\le n}$.*

PROOF. Direct consequence of Proposition 1 and of the fact that a sequence of pushouts is a colimit and a sequence of pullbacks is a limit. □

In the same way that Theorem 1 states the conditions under which the behaviour of a GTS is protected when amalgamated with other GTS, we can get a similar result for multiple amalgamation.

Theorem 2. *Given typed graph transformation systems $GTS_i = (TG_i, P_i, \pi_i)$ and $GTS'_i = (TG'_i, P'_i, \pi'_i)$, for $0 \leq i \leq n$, and $GTS = (TG, P, \pi)$, and given GTS morphisms $f^i\colon GTS_i \to GTS'_i$ and $g^i\colon GTS_i \to GTS$, for $0 \leq i \leq n$, and their multiple amalgamation $\widehat{GTS}$, with induced morphisms $\widehat{f}\colon GTS \to \widehat{GTS}$ and $\widehat{g}^i\colon GTS'_i \to \widehat{GTS}$, for $0 \leq i \leq n$, if each g^i is a behaviour-reflecting GTS morphism, each g^i_{TG} is a monomorphism, and each f^i is an extension and behaviour-protecting morphism, then $\widehat{f}$ is behaviour-protecting and $\widehat{f}_{TG}$ is a monomorphism.*

PROOF. Since by Proposition 7, $GTS\{\oplus_{GTS_i} GTS'_i\}_{0\leq i\leq n} = GTS\{\amalg_{GTS_i} GTS'_i\}_{0\leq i\leq n}$, let us consider the iterative amalgamation, and let us proceed by induction on n.

- If $n = 0$ then this is the case of Theorem 1. Since f^0 is an extension, then f^0_{TG} is a monomorphism, and therefore, since we assume an adhesive HLR category, $\widehat{f}_{TG}$ and $\widehat{g}^0_{TG}$ are monomorphisms.

- Suppose that $n > 0$. (See Figure 6) Let $\widehat{\widehat{GTS}} = GTS\{\oplus_{GTS_i} GTS'_i\}_{0\leq i\leq n-1}$, with morphisms $\widehat{\widehat{f}}\colon GTS \to \widehat{\widehat{GTS}}$ and $\widehat{\widehat{g}}^i\colon GTS'_i \to \widehat{\widehat{GTS}}$, for $0 \leq i \leq n-1$, and let us assume that, because each g^i is a behaviour-reflecting GTS morphism, each g_{TG} is a monomorphism, and each f^i is an extension and behaviour-protecting morphism, that then $\widehat{\widehat{f}}$ is a behaviour-protecting GTS morphism and $\widehat{\widehat{f}}_{TG}$ is a monomorphism.

  By the construction in Definition 17, $\widehat{GTS} = \widehat{\widehat{GTS}} +_{GTS_n} GTS'_n$, and morphisms $\widehat{f}\colon GTS \to \widehat{GTS}$ and $\widehat{g}^i\colon GTS'_i \to \widehat{GTS}$, for $0 \leq i \leq n$, are defined as $\widehat{f} = h \circ \widehat{\widehat{f}}$ and $\widehat{g}^i = h \circ \widehat{\widehat{g}}^i$, for $0 \leq i \leq n-1$, with $\widehat{g}^n\colon GTS'_n \to \widehat{GTS}$ and $h\colon \widehat{\widehat{GTS}} \to \widehat{GTS}$ the morphisms defined by the amalgamation $\widehat{\widehat{GTS}} +_{GTS_n} GTS'_n$ along f^n and $\widehat{\widehat{g}}^n$, with $\widehat{\widehat{g}}^n = \widehat{\widehat{f}} \circ g^n$.

  Since, by the induction hypothesis, $\widehat{\widehat{f}}$ is a behaviour-protecting morphism, it is behaviour-reflecting, and since g^n is also behaviour-reflecting, by Proposition 2, $\widehat{\widehat{g}}^n$ is a behaviour-reflecting GTS morphism. Since g^n_{TG} and $\widehat{\widehat{f}}_{TG}$ are monomorphisms, $\widehat{\widehat{g}}^n_{TG}$ is a monomorphism. Then, since $\widehat{\widehat{g}}^n$ is behaviour-reflecting, $\widehat{\widehat{g}}^n_{TG}$ is a monomorphism and f^n is an extension and behaviour-protecting, then h is behaviour-protecting. Since h and $\widehat{\widehat{f}}$ are behaviour-protecting, then, by Proposition 4, $\widehat{f}$ is also behaviour-protecting. Since h_{TG} and $\widehat{\widehat{f}}_{TG}$ are monomorphisms, then $\widehat{f}_{TG}$ is also a monomorphism.

□

$$\begin{tikzcd}
 & & & GTS_n \arrow[rrdd, "g^n"] \arrow[ddddd, "f^n"] \arrow[rrdddddd, dashed, "\widehat{\widehat{g}}^{\,n}"] & & \\
 & & GTS_{n-1} \arrow[rrrd, "g^{n-1}"] \arrow[ddddd, "f^{n-1}"] & & & \\
 & \iddots & & & & GTS \arrow[dddd, dashed, "\widehat{\widehat{f}}"] \arrow[dddddddd, dotted, bend left, "\widehat{f}"] \\
 & GTS_1 \arrow[rrrru, "g^1"] \arrow[ddddd, "f^1"] & & & & \\
GTS_0 \arrow[rrrrruu, "g^0"] \arrow[ddddd, "f^0"] & & & & & \\
 & & & GTS'_n \arrow[rrddddd, dotted, "\widehat{g}^{\,n}"] & & \\
 & & GTS'_{n-1} \arrow[rrr, dashed, "\widehat{\widehat{g}}^{\,n-1}"] \arrow[rrrdddd, dotted, "\widehat{g}^{\,n-1}"] & & & \widehat{\widehat{GTS}} \arrow[dddd, dotted, "h"] \\
 & \iddots & & & & \\
 & GTS'_1 \arrow[rrrruu, dashed, "\widehat{\widehat{g}}^{\,1}"] \arrow[rrrrdd, dotted, "\widehat{g}^{\,1}"] & & & & \\
GTS'_0 \arrow[rrrrruuu, dashed, "\widehat{\widehat{g}}^{\,0}"] \arrow[rrrrrd, dotted, "\widehat{g}^{\,0}"] & & & & & \\
 & & & & & \widehat{GTS}
\end{tikzcd}$$

Figure 6: Case $n > 2$ in Theorem 2

3. Non-functional properties specification with *e-Motions*

In this section, we show how the theoretical results in the previous sections can be used to safely combine DSLs. Specifically, we will combine DSLs whose syntax is provided by MOF metamodels and whose behaviour is specified by a set of graph transformation rules.

Our first remark is on the kind of graphs we use. The static structure of object-oriented systems is typically modelled with type graphs in which nodes represent classes, and instances of such class diagrams are such that the is-instance relationship is modelled as a typing morphism from the instance to the type graph. Typed attributed graphs [16] support the possibility of defining class attributes, and allocating instance values. Typed attributed graphs with inheritance were introduced in [8, 16] as a way to model the inheritance relation and abstract classes. Polymorphic transformations where introduced to define transformations of such typed attributed graphs with inheritance. A flattening construction was proposed so that such graphs with inheritance have equivalent ones without inheritance, and a polymorphic transformation led to a set of rules typed over the flattened type graph [16]. The category of typed attributed graphs with inheritance was proved to be adhesive HLR in [20]. The approach in [8, 16] was extended in [27] by defining inheritance respecting morphims instead of the flattening construction. The category of typed attributed graphs with inheritance respecting morphisms was proved adhesive in [27]. MOF metamodels may also specify cardinalities, composition relations, etc. Although there are proposals in the literature to deal with some of these features, we have left their inclusion in our setting for future work.

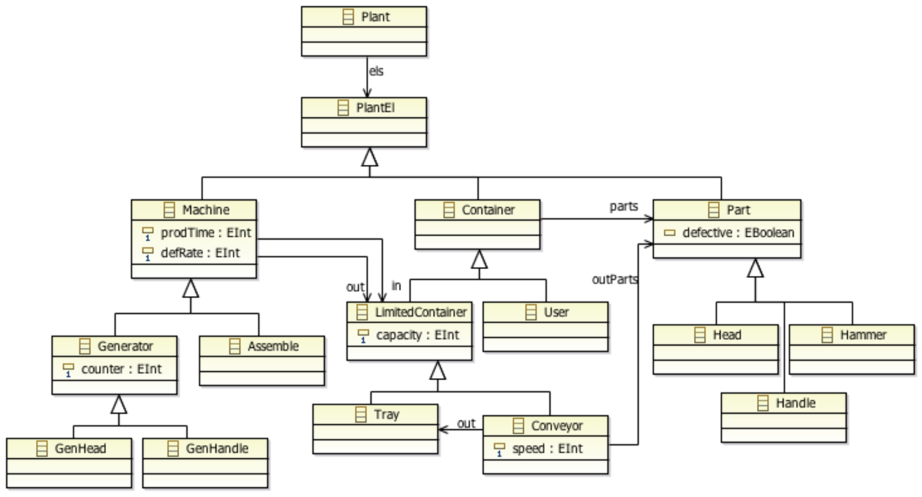
Our motivating example consists of different DSLs which specify some non-functional properties, namely, throughput, response time and failure rate, which are then composed with a DSL of production line systems to be analysed. The resulting DSL will be used to carry out performance analysis on the specified system. Protection of behaviour is key to the usefulness of the results: the behaviour of a model of the production line system described using our DSL will not be affected by the machinery attached to it for its analysis. Our example is adapted and extended from [57]. A proper modelling of time is of course key for such an analysis. Although, as we will see in the following subsections, our transformation rules may use time features, like duration or periodicity, for now we assume that rule morphisms are always between rules with exactly the same time features.

We define our DSLs using the *e-Motions* system [48, 49]. The defined DSLs are then composed using the binding-definition and composition tools described in Section 4, which implement the multi DSL amalgamation construction presented in Section 2. *e-Motions* is a DSL and graphical framework developed for Eclipse that supports the specification, simulation, and formal analysis of DSLs. Given a MOF metamodel (abstract syntax) and a GCS model (a graphical concrete syntax) for it, the behaviour of a DSL is defined by in-place graph transformation rules. *e-Motions* extends standard in-place rules so that time and action statements can be included in the behavioural specifications of a DSL. Although we briefly introduce the language here, we omit all those details not relevant to this paper. We refer the interested reader to [48, 49] or `http://atenea.lcc.uma.es/e-Motions` for additional details.

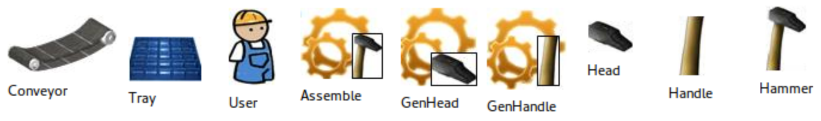
3.1. The Production Line System DSL

Figure 7(a) shows the metamodel of a DSL for specifying Production Line Systems (PLS) for producing hammers out of hammer heads and handles, which are generated in respective machines, transported along the production line via conveyors, and temporarily stored in trays. As usual in MDE-based DSLs, this metamodel defines all the concepts of the language and their interconnections; in short, it provides the language's *abstract* syntax. The diagram in Figure 7(a) is an attributed type graph with inheritance, describing the types and their hierarchies to be used for the typing of the instances. More specifically, an attributed type graph with inheritance (ATG, I, A) consists of an attributed type graph ATG defining the types and their composition/association relations, an inheritance graph I with the same nodes as ATG, and a subset A of them with the abstract nodes.

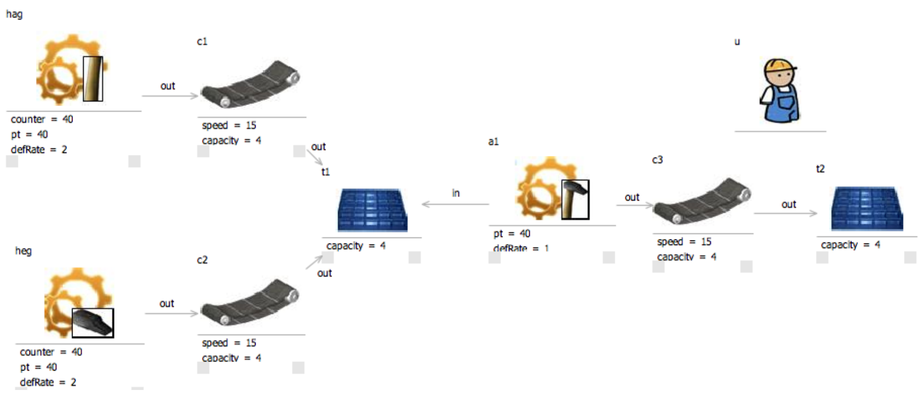
A *concrete* syntax is provided as a Graphical Concrete Syntax (GCS) model. In the case of our example, this is sufficiently well defined by providing icons for each concept (see Figure 7(b)); connections between concepts will then be indicated through



(a) PLS metamodel



(b) PLS concrete syntax



(c) Example of production line configuration

Figure 7: Production Line System metamodel (a) and concrete syntax (b)

arrows connecting the corresponding icons. Figure 7(c) shows a model conforming to the metamodel in Figure 7(a) using the graphical notation introduced in the GCS model in Figure 7(b).

The behavioural semantics of the DSL is then given by providing transformation rules specifying how models can evolve. Figure 8 shows an example of such a rule. The rule consists of a left-hand side matching a situation before the execution of the

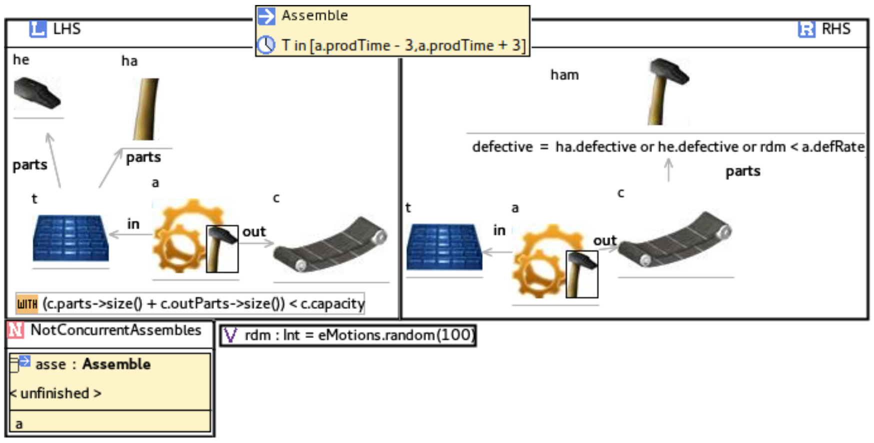

Assemble
T in [a.prodTime - 3,a.prodTime + 3]
LHS
RHS
he
ha
parts
parts
a
t
in
out
c
(c.parts->size() + c.outParts->size()) < c.capacity
ham
defective = ha.defective or he.defective or rdm < a.defRate
parts
NotConcurrentAssembles
asse : Assemble
< unfinished >
a
rdm : Int = eMotions.random(100)

Figure 8: The Assemble rule indicates how a hammer is assembled

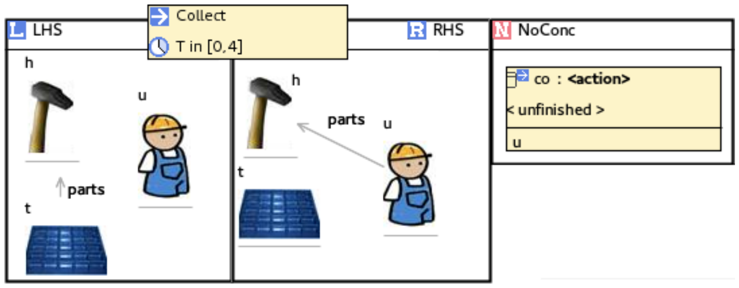


Figure 9: The Collect rule specifies how a hammer is removed from the system

rule and a right-hand side showing the result of applying the rule. Specifically, this rule shows how a new hammer is assembled: a hammer generator a has an incoming tray of parts and is connected to an outgoing conveyor belt. Whenever there is a handle and a head available, and there is space in the conveyor for at least one part, the hammer generator can assemble them into a hammer. The new hammer is added to the parts set of the outgoing conveyor belt in time T, with T some value in the range [a.pt-3, a.pt+3], and where pt is an attribute representing the production time of a machine. Notice the use of both positive and negative conditions constraining the matches of this rule. In *e-Motions*, we may have as many positive and negative conditions as necessary, and we can even have positive conditions on NACs (nested application conditions). Each action is represented during execution by an *action* object, which we may use as any other object. By having an Assemble action object in its NAC, we ensure that an assemble action is not initiated if there is another one under execution, that is, we explicitly indicate that the assemble machine is not in use.

Figure 9 shows another of the rules specifying the behaviour of the PLS. This rule represents the action in which a worker takes a hammer from the final tray and removes it from the system. Notice that the duration of the action is some value between 0 and 4.

The Assemble and Collect rules are *atomic* rules. In e-Motions, there are two types of rules to specify time-dependent behaviour, namely, *atomic* and *ongoing* rules. Atomic rules represent atomic actions with a duration, which is specified by an interval of time. Atomic rules with duration zero are called *instantaneous* rules. Ongoing rules represent actions that progress continuously over time and that can be interrupted at any time, while the rule's preconditions (LHS and not NACs) hold. Atomic rules can be periodic, and both atomic and ongoing rules can be scheduled, or be given an execution interval.

The complete behaviour of our PLS DSL is defined by a number of such rules that specify the different actions that can take place in the system, e.g., generating new pieces, moving pieces from a conveyor to a tray, etc. Its complete specification using *e-Motions* can be found at `http://atenea.lcc.uma.es/E-motions/PLSExample`.

3.2. Parametric DSLs for the independent definition of non-functional properties

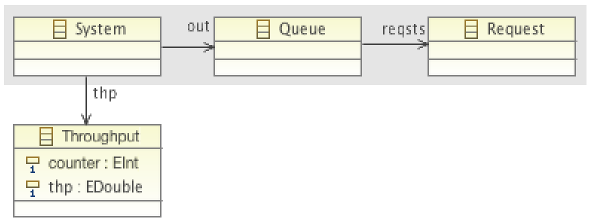
In this section we model a number of different non-functional properties, namely throughput, response time, and failure rate. Clearly, these properties could be useful in relation to other systems as well, so we will first define and model them independently of the PLS DSL. Following [57, 58], we rely on the notion of observer to analyse non-functional properties of systems described by GTSs in a way that can be analysed by simulation. By specifying different properties to be analysed as separate, parameterized DSLs, independent of the definition of any system, we are then able to compose these DSLs with the base DSL, the PLS DSL in this case, to generate specific simulation environments.

3.2.1. Throughput

In communication networks, *throughput* is defined as the average rate of message delivery over a communication channel. However, the notion has also been used in other disciplines, like costing and manufacturing, acquiring a more general meaning. We can define throughput as the average rate of work *items* flowing through a system. Thus, the same generic notion allows us to measure the number of information packages being delivered through a network, the number of passengers checking-in in an airport desk, or the number of hammers being manufactured in a production line.

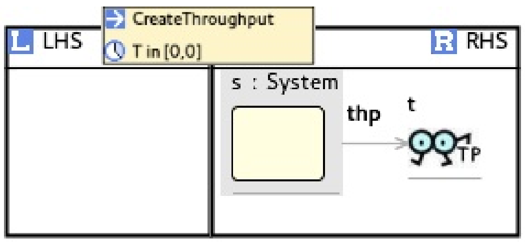
Given this more general definition, and given the description of a system, to measure throughput, we basically need to be able to count the number of items delivered or produced, and calculate its quotient with time. We define a ThroughputOb observer class with attributes counter and thp keeping these values. The metamodel for the DSL of ThroughputOb observers is the one depicted in Figure 10(a). ThroughputOb observer objects will basically count instances of some *generic* class Request, which, as we will see in Section 3.3, will later be instantiated to parts, as could be instantiated to data packages or to passengers. These ThroughputOb objects will be associated to specific systems, so that we may, e.g., measure the throughput of each of the connections in a network, each of the check-in desks in an airport, or each of the production lines in a factory.

Given a concrete syntax, depicted in Figure 10(b), the behaviour of ThroughputOb objects is defined by the transformation rules CreateThroughput, RecordLeave and

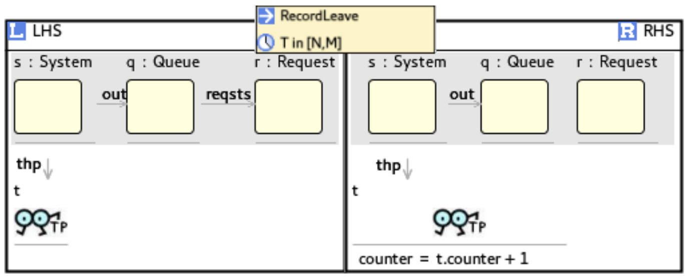


(a) Abstract syntax

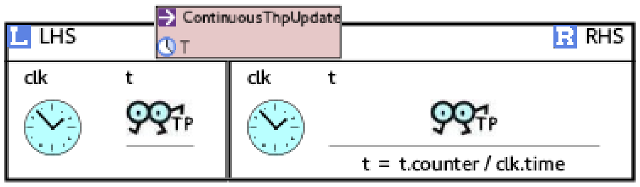
(b) Concrete syntax



(c) CreateThroughput atomic rule



(d) RecordLeave atomic rule



(e) ContinuousThpUpdate on-going rule

Figure 10: Throughput observer DSL definition

ContinuousThpUpdate, shown in Figures 10(c), 10(d) and 10(e), respectively. The CreateThroughput rule specifies how throughput observer objects are created. The RecordLeave rule represents the way in which the throughput observer counts processed requests, that is, it represents the way in which the values of its counter attribute is to be updated: when a request leaves the System's out queue, the ThroughputOb observer gets updated. The ContinuousThpUpdate rule is in charge of maintaining the actual throughput value updated. Note that whilst rules CreateThroughput and RecordLeave, represent atomic actions, ContinuousThpUpdate is an on-going rule, ensuring that the thp attribute is always updated. Clock is a predefined class in *e-Motions* which represents the passage of time in the system, and whose time attribute keeps the time of the system since its start-up.

Since classes System, Server and Request are generic, no concrete syntax is provided for them, and therefore instances of them are represented as boxes. Note that this DSL is not usable by itself, it is a generic DSL — the parameter part of generic DSLs is depicted shadowed in Figure 10 and in all other generic DSL definitions — and needs to be instantiated before used. The non-shadowed part of a rule describes the extensions that are required. So, in addition to reading, e.g., Figure 10(d) as a 'normal' transformation rule (as we have done above), we can also read it as a *rule transformation*, stating: "Find all rules that match the shaded pattern and add Throughput objects to their left- and right-hand sides as described." In effect, observer models become higher-order transformations [56] in our implementation (see Section 4).

3.2.2. Response time

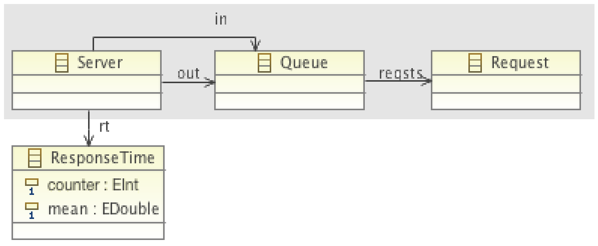
Like throughput, response time is also a property we may want to observe on many different systems. We may want to know the time taken by a web server to answer a request, how long a data package takes to go from one node to another in a network, how long it takes to check-in in an airport, or how long a hammer takes to be produced. Figure 11(a) shows the metamodel for a DSL for specifying response time. It is defined as a parametric model (i.e., a model template). It uses the notion of response time, which can be applied to different systems with different meanings. The concepts of Server, Queue, and Request and their interconnections are parameters of the metamodel (and therefore shaded in grey). Figure 11(b) shows the concrete syntax for the response time observer object.

Figures 11(c) and 11(d) show the transformation rules defining the behaviour of the response time observer. Rule CreateRT specifies the creation of response-time observer objects. The ResponseTime rule states that if there is a server with an in queue and an out queue and there initially are some requests (at least one) in the in queue, and the out queue contains some requests after rule execution, the last response time should be recorded to have been equal to the time it took the rule to execute.

In the rule in Figure 11(d), we use a cardinality constraint $1..*$ to indicate that the actual parameter with which we instantiate the DSL will follow this *pattern*; that is, it is just syntactic sugar to cover different alternatives. According to our formalisation in Section 2, the morphism from the parameter DSL to the system DSL will have to map n Request objects [r1] and n reqsts links and m Request objects [r2] and m links to corresponding n and m objects and links in the target DSL (notice that the number of objects and links for the left and right-hand sides of a rule may be different). Once the matching indicates the values of these n's, the construction only has to deal with that specific matching.

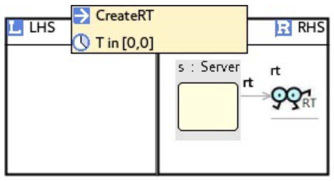
3.2.3. Failure rate

The failure rate property may be defined similarly. Figure 12 shows the failure rate observer DSL definition. As for the response time and throughput ones, the metamodel

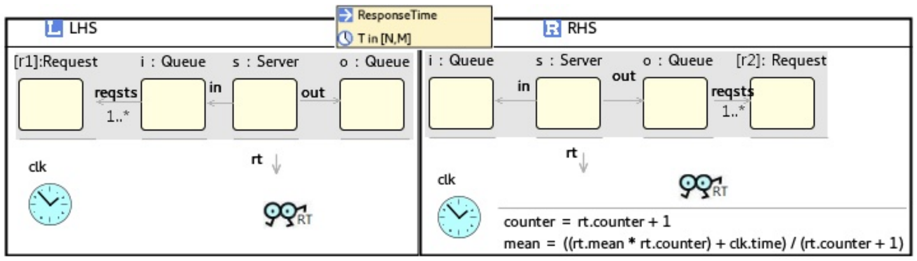


(a) Metamodel

(b) Concrete syntax



(c) CreateRT atomic rule

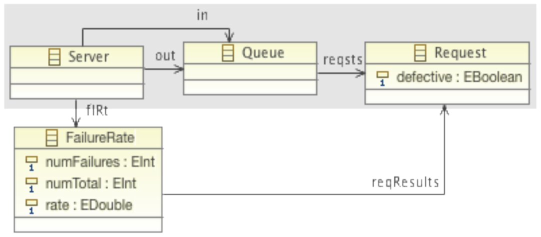


(d) ResponseTime atomic rule

Figure 11: Generic model of response time observer

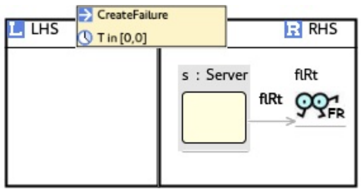
for the failure rate DSL is defined on a parameter DSL that includes those generic elements it depends on. In this case, the metamodel also specifies a FailureRate class with attributes numFailures, to count failures, numTotal, to keep the total number of processed elements, and rate, in which the actual failure rate is stored. Notice the reqResults link, which will be used to temporarily collect all processed requests.

The behaviour of the DSL is then specified by rules CreateFailure (Figure 12(c)), which creates observer objects, FailureRateSeq (Figure 12(d)), which adds every processed request to the reqResults collection, and FailureRateCalc1 (Figure 12(e)) and FailureRateCalc2 (Figure 12(f)), which process the requests in the reqResults collection and update the values of the corresponding attributes. Whilst the FailureRateSeq rule is atomic, FailureRateCalc1 and FailureRateCalc2 are on-going rules, what ensures that the failure rate will be always updated.

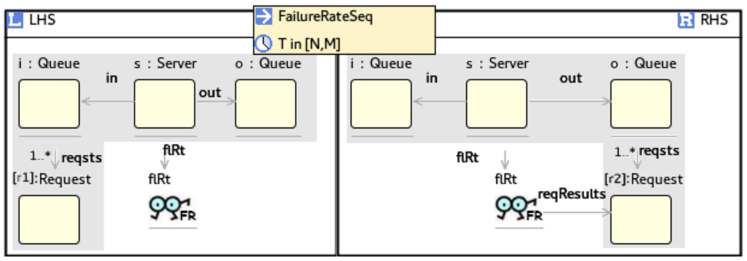


(a) Abstract syntax

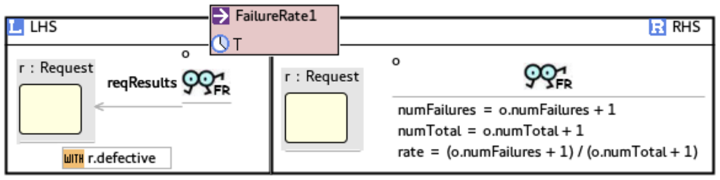
(b) Concrete syntax



(c) CreateFailureRate atomic rule

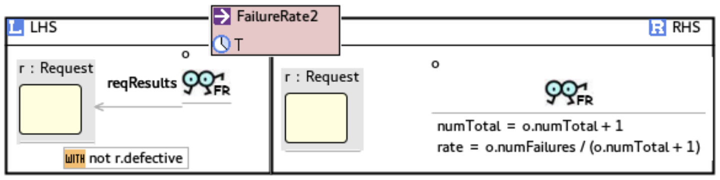


(d) FailureRateSeq atomic rule



LHS
FailureRate1
T
RHS
o
r : Request
reqResults
FR
WITH r.defective
r : Request
o
FR
numFailures = o.numFailures + 1
numTotal = o.numTotal + 1
rate = (o.numFailures + 1) / (o.numTotal + 1)

(e) FailureRateCalc1 on-going rule



(f) FailureRateCalc2 on-going rule

Figure 12: Failure rate observer DSL definition

$$\begin{tikzcd}
\boxed{\begin{array}{l} M_{Par} \\ \quad MM_{Par} \oplus Rls_{Par} \end{array}} \arrow[r, Rightarrow, "\begin{array}{c}\mathit{Binding} \\ B_{MM} \oplus B_{Rls}\end{array}"] \arrow[d, Rightarrow] & \boxed{\begin{array}{l} M_{DSL} \\ \quad MM_{DSL} \oplus Rls_{DSL} \end{array}} \arrow[d, dotted] \\
\boxed{\begin{array}{l} M_{Obs} \\ \quad MM_{Obs} \oplus Rls_{Obs} \end{array}} \arrow[r, dotted] & \boxed{\begin{array}{l} M_{\widehat{DSL}} \\ \quad (MM_{DSL} \otimes MM_{Obs}) \oplus (Rls_{DSL} \otimes Rls_{Obs}) \end{array}}
\end{tikzcd}$$

Figure 13: Amalgamation of DSLs

3.3. DSL weaving for analysis

To be able to collect information on the production time of parts, number of parts produced per time unit, and rate of defective parts in the PLS DSL described in Section 3.1, we need to weave the languages together. In fact, since we could want, e.g., to measure the response time of each of the machines in a production line, or the number of defective hammer heads or hammer handles, we may want to compose the PLS DSL with more than one instance of some of our generic DSLs. We may do this just by providing appropriate binding morphisms.

According to Definition 6, a GTS morphism is given by a morphism between the underlying type graphs, a surjective mapping between the sets of rule names, and a family of rule morphisms. In terms of DSLs, whose semantics is given by corresponding GTSs, given the amalgamation construction in Definition 15, if we want to compose, e.g., the response time observer DSL with the PLS DSL to measure the production time of hammers, we need to provide GTS morphisms from the parameter sub-DSL of the response time DSL to the response time DSL itself, and a binding morphism between this parameter sub-DSL and the PLS DSL. The weaving of DSLs then corresponds to amalgamation in the category of GTSs and GTS morphisms (Definition 15). Figure 13 shows the amalgamation of an inclusion morphism between the model of an observer DSL, M_{Obs} (abstract syntax given by a metamodel MM_{Obs} and behaviour given by a set of transformation rules Rls_{Obs}), and its parameter sub-model M_{Par} ($MM_{Par} \oplus Rls_{Par}$), and the binding morphism from M_{Par} to the DSL of the system at hand, M_{DSL}, the PLS DSL in our example. The amalgamation object $M_{\widehat{DSL}}$ is obtained by the construction of the pushout of the corresponding metamodel morphisms and the amalgamation of the rules describing the behaviour of the different DSLs.

Thus, to, e.g., collect information on the production time of hammers in our PLS DSL, we need to provide a binding from the elements in the parameter part of the observer DSL metamodel (e.g., Figure 11(a)) to elements in the PLS metamodel (Figure 7(a)). We start with the response-time DSL and bind it so as to measure the response time of the Assemble machine:

- Classes:
  - Server to Assemble;
  - Queue to LimitedContainer, as the Assemble machine is to be connected to an arbitrary LimitedContainer for queuing incoming and outgoing parts; and

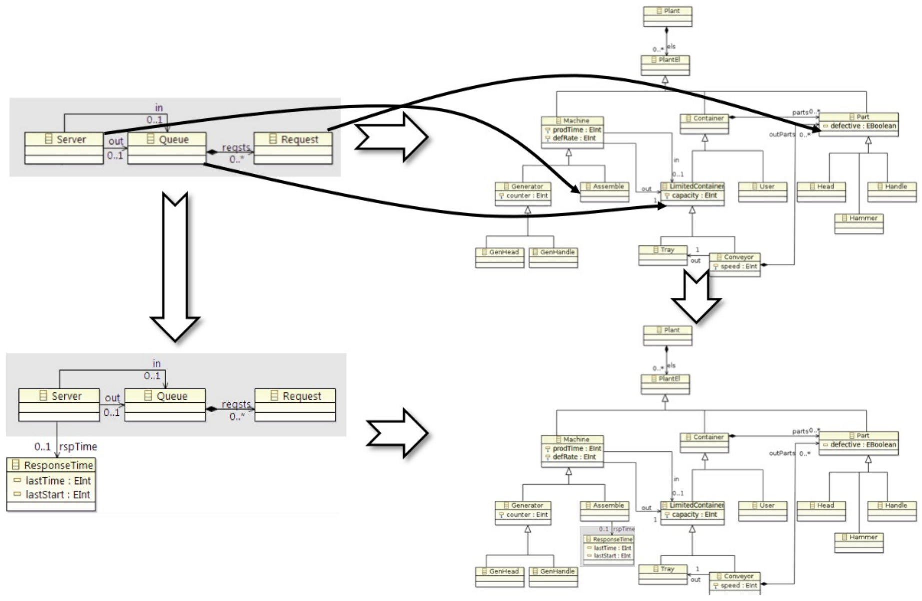


Figure 14: Weaving of metamodels

- Request to Part, as Assemble only does something when there are Parts to be processed; and

- Associations:
  - The in and out associations from Server to Queue are bound to the corresponding in and out associations from Machine to LimitedContainer, respectively; and
  - The association requests from Queue to Request is bound to the association parts from Container to Part.

This morphism between the attributed type graphs with inheritance has its correspondent one over the flattened versions of the graphs.

Figure 14 illustrates the composition of the metamodels (pushout of the corresponding type graphs). We observe how the weaving process adds the ResponseTime concept to the resulting metamodel. Notice that the weaving process also ensures that only sensible woven metamodels can be produced: for a given binding of parameters, there needs to be a match between the requirements expressed in the observer metamodel (associations, cardinalities, etc.) and the DSL metamodel.

The binding is completed with maps for rules. Each rule in the parameter DSL is mapped to a rule in the target DSL. In the formalisation in Section 2, see Definition 6, this binding includes both a rule-name mapping and a set of rule morphisms. The intuition is that each rule in the parameter DSL must be mapped to a rule in the target DSL. A rule morphism is assumed to be defined between rules with the same time features, and must specify to which element in the target rule each of the elements

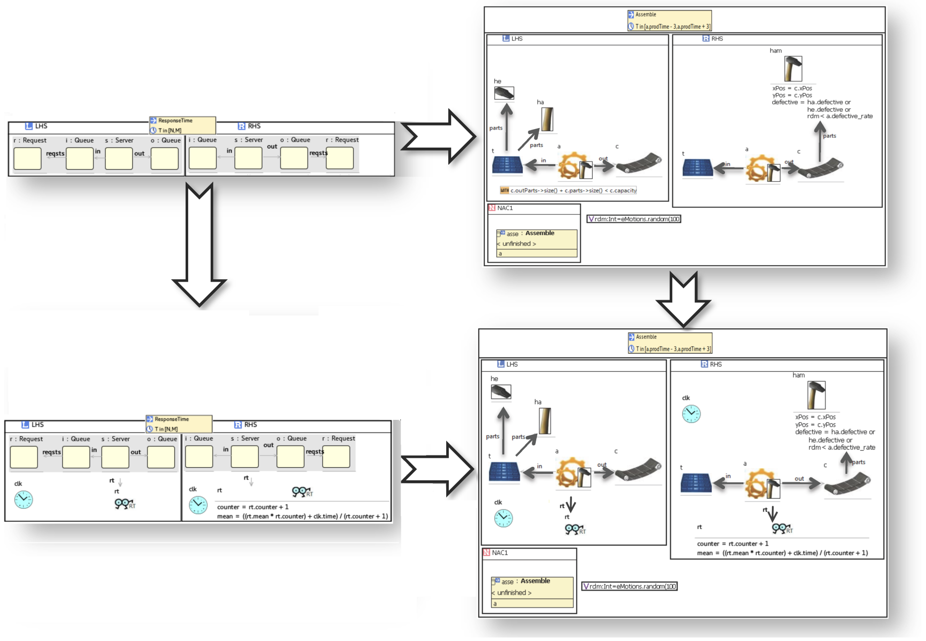


Figure 15: Amalgamation of the Assemble and RespTime rules

in the source rule is sent. For example, the rule Assemble in Figure 8 matches the pattern in Figure 11(d), given this binding: In the left-hand side, there is a Server (Assemble) with an in-Queue (Tray) that holds two Requests (Handle and Head) and an out-Queue (Conveyor). In the right-hand side, there is a Server (Assemble) with an in-Queue (Tray) and an out-Queue (Conveyor) that holds one Request (Hammer). Figure 15 shows the rule amalgamation using this morphism. We can see how the obtained amalgamated rule is similar to the Assemble rule but with the observers in the RespTime rule appropriately introduced.

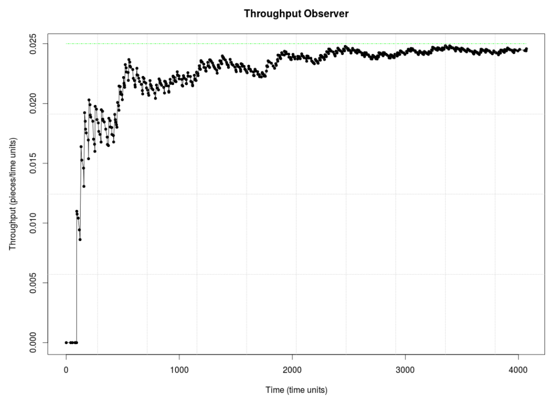
The DSL resulting from such amalgamation will have a metamodel extended with observer classes, and its rules will be decorated with observer objects newly introduced, or ready to collect information on the execution. This composed DSL may also have additional rules, resulting from rules in the observer DSL which do not change parameter objects. This is the case of rule ContinuousThpUpdate (Figure 10(e)) in the Throughput observer DSL, or rules FailureRateCal1 and FailureRateCal2 (Figures 12(e) and 12(f)) in the Failure rate observer DSL. Note that the composition results in a DSL completely operational, the DSLs involved have been completely instantiated and can be executed.

Once appropriate binding morphisms for the throughput and failure rate observers DSLs are defined in a similar way, we may use the multiple amalgamation construction to compose all of them in one single DSL. Given a binding for the throughput to compute the throughput on the entire production line, and calculate the failure rate of the assemble machine, we get a 'decorated' production line system on which we may observe all these properties. Figure 16 shows the data collected along time during the execution of the system resulting from the amalgamation for a run of 4000 time units. Figure 16(a) shows how the throughput of the machine Assemble evolves along time. Such a chart illustrates how the throughput grows from zero to the asymptote, as time passes. Figure 16(b) shows the mean failure rate of the same machine along time. Gaps in the chart represent failures in the machine. Finally, Figure 16(c) shows the mean response time of the Assemble machine along time.

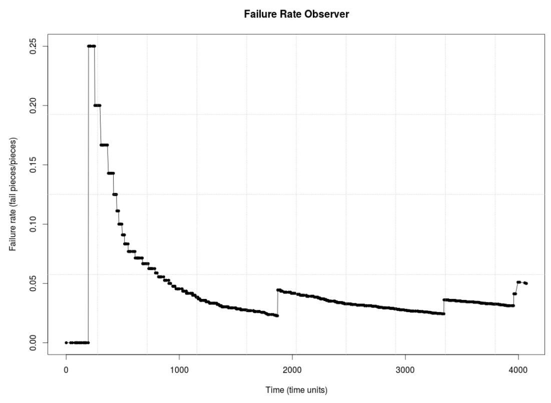
The amalgamation of the different DSLs has allowed us to analyse the performance of the system resulting from the amalgamation. However, we still need to check the conditions of Theorem 2, so that we can infer that the behaviour of the original system did not change, and therefore, that the analysis results are indeed for the original system. We have to check that (1) the inclusion morphisms between the parameter DSLs and their corresponding observer DSLs are extensions and behaviour protecting, and (2) that the morphisms between such parameter DSLs and the PLS DSL are behaviour-reflecting and their underlying metamodel morphisms are monomorphisms.

In this concrete simple case study the checks can be carried out by hand without much effort. For parameterized DSLs, we will always have that the parameter DSL is included in the DSL it is a parameter of. If we look at the observer parametric DSLs defined in Section 3.2, we easily check that they are extensions: all the additions in the rules use properties not in the parameter, that is, features that would be removed if moving back along the inclusion morphisms (using the corresponding backward retyping functors). By Lemma 1, extensions are behaviour-reflecting, so we are left with checking that they preserve behaviour and deadlocks. Traces in the parameter DSL of the throughput observer DSL (shadowed part of Figure 10) will be sequences of CreateThroughput and RecordLeave rules. The former can always be executed, both if considered at the parameter DSL or at the observer DSL. If the RecordLeave rule can be applied for the parameter DSL, it will also be applied for the observer DSL, perhaps after an additional application of the CreateThroughput rule if the System object had no associated thp observer yet (stuttering steps). Deadlocks are also preserved. Indeed, the parameter DSL is deadlock free, since the CreateThroughput rule can always be applied. A similar reasoning may be carried out for the other parametric DSLs in Section 3.2.

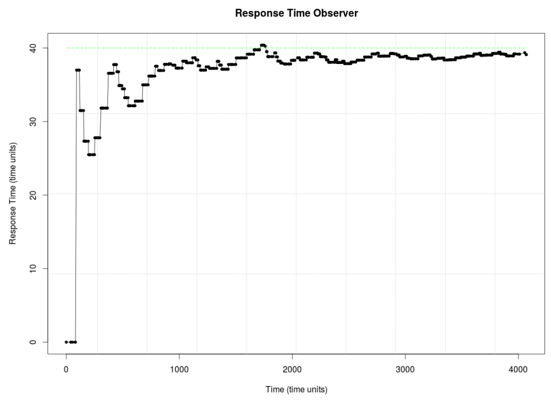
Regarding the morphisms from the parameters of the observer DSLs and the PLS DSL, we need to check that their underlying metalevel morphisms are monomorphims, and that they reflect behaviour. Checking that they are monomorphisms is straightforward. Checking that they reflect behaviour is non-trivial, but if we study the morphisms defined in this case, we realize that they actually are extensions. Notice how similar are the structures of the parameter part of rule RecordLeave (Figure 10(d)) and rule Collect (Figure 9), and the parameter part of rule ResponseTime (Figure 11(d)) and rule Assemble (Figure 8). The same occur for the other rule morphisms. Since extensions are behaviour-reflecting (by Lemma 1) we have completed the checks of all required conditions, and we can therefore conclude that the behaviour of the PLS DSL did not change when amalgamated with its observers.



(a) Throughput of the production line



(b) Mean failure rate of the Assemble machine



(c) Mean response time of the Assemble machine

Figure 16: Analysis results

4. Implementation: Binding definitions and iterative amalgamation

We have implemented the composition operations presented in Section 2. Specifically, we have extended the implementation presented in [14] and added support for multiple amalgamation. Our Eclipse plugin enables language developers to provide a DSL and a set of parametric DSLs with their parameter bindings; the plugin then allows the DSL weaving to be triggered. DSL weaving is implemented using an extended version of the ATL transformation discussed in [14].

4.1. Binding algorithm

Manually defining the binding between parametric and base DSL can be quite cumbersome and error prone. To ease the process, we have implemented an algorithm that semi-automatically proposes valid bindings. We currently support incremental semi-automatic binding for meta-models, but not for rules.

The meta-model part of the parameter DSL encodes the structure of meta-classes and meta- associations that need to be matched in the base DSL for a valid binding to be established. For example, in our case study, class Server of the *response time* meta-model can be bound only with certain classes of the *production line* meta-model, namely with classes that have two references to a class bound to Queue and one class bound to Request. By appropriately encoding these constraints in Maude [5], we have transformed the problem into a matching problem, which can be automatically solved to retrieve the complete set of all valid bindings. As long as more than one binding is found, the user is asked to provide additional partial binding information (for example, by binding one meta-class in the parameter DSL) and the constraint solver is run again. When only one valid solution remains, this is directly proposed to the user. Once the algorithm has been executed through the Eclipse plug-in, we get a list of possible substitution, each of the form:

```
Server <- assemble,
Queue <- limitedContainer,
...
```

This is only part of the binding, we also need to bind the rules. We believe that the binding of the behavioural part may be inferred semi-automatically, too. For example, let us suppose that parametric class Par_A and actual class Act_A match, then object o_parA of type Par_A is going to match with object o_actA of type Act_A. We will study this approach further as part of our future work.

4.2. Transformations

We have implemented model-to-model transformations to compose both syntax and behaviour of DSLs. These transformations have been coded using ATL [36]. We have split the composition process into two different ATL transformations: one for weaving the syntax, both abstract and concrete syntaxes, i.e., MM_{DSL} and MM_{Obs}; and another one for weaving the behavioural rules, i.e. Rls_{DSL} and Rls_{Obs}. The former produces the woven syntax of our DSL (for example, the meta-model obtained in Figure 14) while the latter produces the woven behaviour of our DSL (the rule obtained in Figure 15).

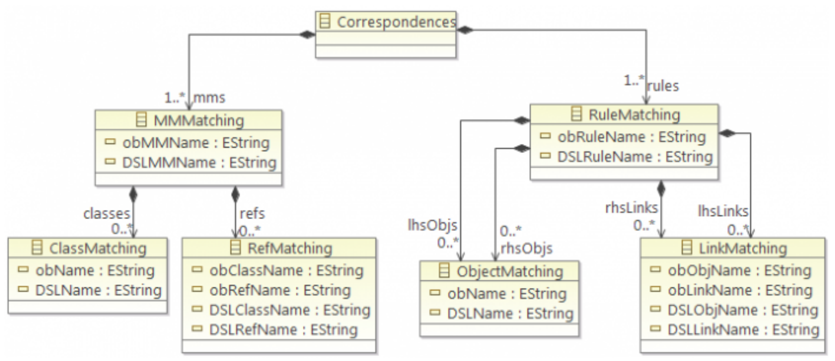


Figure 17: Correspondences meta-model

Bindings — B_{MM} and B_{Rls} — are represented in our implementation by a model, conforming to the *correspondences meta-model* shown in Figure 17. Each individual binding (be it between meta-classes or between rules and their parts) is represented by its own model element. This meta-model is similar to the weaving models introduced, for example, in the context of ATL [24]. The completed correspondences model is used as input for both model transformations.

The *e-Motions* models obtained as result of the weaving can be automatically transformed into Maude specifications (see [49] for details on this transformation) and further analysed using standard Maude tooling. See [47] for a detailed presentation of how Maude provides an accurate way of specifying both the abstract syntax and the behavioural semantics of models and meta-models, and offers good tool support both for simulating and for reasoning about them.

5. Related Work

Graph transformation systems (GTSs) were proposed as a formal specification technique for the rule-based specification of the dynamic behaviour of systems [15]. Different approaches exist for modularisation in the context of the graph-grammar formalism [6, 52, 16]. All of them have followed the tradition of modules inspired by the notion of algebraic specification module [19]. A module is thus typically considered as given by an export and an import interface, and an implementation body that realizes what is offered in the export interface, using the specification to be imported from other modules via the import interface. For example, Große-Rhode, Parisi-Presicce, and Simeoni introduce in [29] a notion of *module* for typed graph transformation systems, with interfaces and implementation bodies; they propose operations for union, composition, and refinement of modules. Other approaches to modularisation of graph transformation systems include PROGRES Packages [54], GRACE Graph Transformation Units and Modules [39], and DIEGO Modules [55]. See [34] for a discussion on these proposals. For the kind of systems we deal with, the type of module we need is much simpler. For us, a module is just the specification of a system, a GTS, without import and export interfaces. Then, we build on GTS morphisms to compose these modules, and specifically we define parameterized GTSs.

We find different forms of GTS morphisms in the literature, taking one form or another depending on their concrete application. Thus, we find proposals centered on refinements [33, 28, 29], views [23], and substitutability [22]. See [22] for a first attempt to a systematic comparison of the different proposals and notations. None of these notions fit our needs, and none of them coincide with our behaviour-aware GTS morphisms.

As far as we know, parameterized GTSs and GTS morphisms, as we discuss them, have not been studied before. Heckel and Cherchago introduce parameterized GTSs in [32], but their notion has little to do with our parameterized GTSs. In their case, the parameter is a signature, intended to match service descriptions. They however use a double-pullback semantics, and have a notion of substitution morphism which is related to our behaviour-preserving morphism.

The way in which we think about composition of reusable DSL modules is related to work in aspect-oriented modeling (AOM). In particular, our ideas for expressing parameterized metamodels are based on the proposals in [4, 38]. Most AOM approaches use syntactic notions to automate the establishment of mappings between different models to be composed, often focusing primarily on the structural parts of a model. While our mapping specifications are syntactic in nature, we focus on composition of behaviours and provide semantic guarantees. In this sense, our work is perhaps most closely related to the work on MATA [60] or semantic-based weaving of scenarios [37]. Although the approach is completely different to ours, Machado, Foss and Ribeiro propose in [41] an extension to conventional graph grammar, which they call aspect-oriented graph grammars, where aspects are modeled as transformation rules over the structure of a base graph grammar.

Genericity has also been used by different authors to enable reuse of model-management operations across different DSLs. For example, [51, 9] use generic metamodel *concepts* as an intermediate, abstract metamodel over which model management specifications are defined, enabling the application of the operations thus defined to any metamodel satisfying the requirements imposed by the concept. Similarly, [30] introduces a notion of meta-model sub-typing to achieve a comparable effect. A comparison of these approaches and a first attempt to synthesise them has been reported in [62]. Figure 18 demonstrates how these approaches fit into our **GTS** category. It can be seen that both model typing and model concepts represent pathological cases where the parameter DSL MT and the base DSL MM do not have any associated operational semantics. Model concepts use explicit bindings between MT and MM and, given a transformation $XForm$ defined on MT, perform a translation of $XForm$ to transformation $\widehat{XForm}$ defined on MM, while model typing uses default bindings and backward typing to execute transformations.

6. Conclusions

We have proposed in this paper a novel concept of parameterised graph transformation system and their instantiation by amalgamation. Such amalgamations are defined

$$\begin{tikzcd}
MT \arrow[r, Rightarrow, "B"] \arrow[d, Rightarrow, "i"'] & MM \arrow[d, dotted, "\widehat{i}"'] \\
MT \otimes XForm \arrow[r, dotted, "\widehat{B}"] & MM \otimes \widehat{XForm}
\end{tikzcd}$$

Figure 18: Model concepts and model typing in **GTS**

for multiple parameters, and we have proven that the multiple amalgamation is in fact equivalent to its iterative version.

We have built this construction on a novel definition of behaviour-aware morphisms of systems. Specifically, we introduce behaviour-reflecting, -preserving, and -protecting morphisms. Some of these novel definitions rely on ideas from concurrency theory (stuttering bisimulations) to reason on the behaviours of system as given by all its possible traces. Given these notions, we are able to prove that the instantiation of systems satisfying certain properties do not change their behaviour.

The approach is illustrated by the use of observers to carry out performance analysis of system. Indeed, our work was originally motivated by the specification of non-functional properties (NFPs), such as performance or throughput, in DSLs. We have been looking for ways in which to encapsulate the ability to specify non-functional properties into reusable DSL modules. Troya et al. used the concept of observers in [57, 58] to model non-functional properties of systems described by GTSs in a way that could be analysed by simulation. Zschaler had previously discussed modular specification of NFPs in temporal logic in [61]. In [14, 58], we have combined these ideas to allow the modular encapsulation of observer definitions in a way that can be reused in different DSL specifications. A first attempt to formalise the needed composition operations was provided in [13], were we provided a formal framework of such language extensions. In [44], we addressed the performance analysis problem by presenting a model-based and modular partial reimplementation of one well-known analysis framework — the Palladio Architecture Simulator. We have specified key DSLs from Palladio in e-Motions, describing the basic simulation semantics as a set of graph-transformation rules. Different properties to be analysed have been encoded as separate, parameterized DSLs, independent of the definition of Palladio. We have then composed these DSLs with the base Palladio DSL to generate specific simulation environments. Models created in the Palladio IDE can be fed directly into our simulation environment for analysis.

This paper extends these earlier works in three main ways:

1. We provide the formal tools needed to seamlessly extend our concepts to the composition of more than two DSLs;
2. We provide a revised notion of behaviour preservation that can be shown to extend to sequences of transformation steps in a GTS rather than only individual rules; and
3. An implementation of the multiple amalgamation construction has been provided.

Much work still remains. The three main areas of future work for us are:

1. *Improving the flexibility of DSL composition.* Currently, our formal framework only supports morphisms (and by extension amalgamations) where there is a close correspondence between the set of transformation rules in the GTSs to be composed. This limits the amount of reuse that can be achieved. We plan to study ways in which the relationships can be made more flexible — for example, by allowing GTS morphisms to selectively map rules or map individual rules to sequences of rules. Similarly, type-graph morphisms currently require a very high structural similarity between the two type graphs. We are planning to explore bindings inspired by type groups [2] and ideas by de Lara and Guerra in [10] to allow for more flexibility in this area.
2. *Improving the tool support.* There is reason to be optimistic that at least a substantial set of behaviour-preserving morphisms can be verified statically by establishing simple, easily checkable conditions over rules and sets of rules. We plan to provide such conditions and embed them into our current tooling so that behaviour-protection of any DSL composition can be verified effectively.
3. *Exploring other DSL composition scenarios.* So far, we have focused on scenarios where we are adding observer DSLs to a base DSL. For these cases, behaviour-protection is an appropriate property to require of the underlying morphisms. However, other DSL-composition scenarios may find this property too restrictive. We plan to identify classes of DSL-composition scenarios that can be characterised by different semantics-related properties of the underlying morphisms.

Acknowledgements

We would like to thank the anonymous reviewers for their careful reading of our work and the comments made, which we found very useful in order to improve the quality and clarity of our article. The work of Fernando Orejas has been partially supported by funds from the Spanish Ministry for Economy and Competitiveness (MINECO) and the European Union (FEDER funds) under grant COMMAS (ref. TIN2013-46181-C2-1-R).

References

[1] P. Boehm, H.-R. Fonio, and A. Habel. Amalgamation of graph transformations with applications to synchronization. In H. Ehrig, C. Floyd, M. Nivat, and J. W. Thatcher, editors, *TAPSOFT, Vol.1*, volume 185 of *Lecture Notes in Computer Science*, pages 267–283. Springer, 1985.

[2] K. B. Bruce, L. Petersen, and A. Fiech. Subtyping is not a good "match" for object-oriented languages. In M. Akşit and S. Matsuoka, editors, *Proc. 11th European Conf. on Object-Oriented Programming (ECOOP'97)*, volume 1241 of *LNCS*, pages 104–127. Springer, 1997.

[3] K. Chen, J. Sztipanovits, S. Abdelwalhed, and E. Jackson. Semantic anchoring with model transformations. In *Proc. of the 1st European Conference on*

Model Driven Architecture: Fondations and Applications (ECMDA-FA'05), volume 3748 of *Lecture Notes in Computer Science*. Springer, 2005.

[4] S. Clarke and R. J. Walker. Generic aspect-oriented design with Theme/UML. In *Aspect-Oriented Software Development*, pages 425–458. Addison-Wesley, 2005.

[5] M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and C. L. Talcott. *All About Maude*, volume 4350 of *LNCS*. Springer, 2007.

[6] A. Corradini, H. Ehrig, M. Löwe, U. Montanari, and J. Padberg. The category of typed graph grammars and its adjunctions with categories of derivations. In Cuny et al. [7], pages 56–74.

[7] J. Cuny, H. Ehrig, G. Engels, and G. Rozenberg, editors. *Proc. 5th International Workshop on Graph Grammars and their Applications to Computer Science (GraGra 1994)*, volume 1073 of *Lecture Notes in Computer Science*. Springer, 1996.

[8] J. de Lara, R. Bardohl, H. Ehrig, K. Ehrig, U. Prange, and G. Taentzer. Attributed graph transformation with node type inheritance. *Theoretical Computer Science*, 376(3):139–163, 2007.

[9] J. de Lara and E. Guerra. From types to type requirements: genericity for model-driven engineering. *Software and System Modeling*, 12(3):453–474, 2013.

[10] J. de Lara and E. Guerra. Towards the flexible reuse of model transformations: A formal approach based on graph transformation. *J. Log. Algebr. Meth. Program.*, 83(5-6):427–458, 2014.

[11] J. de Lara and H. Vangheluwe. Automating the transformation-based analysis of visual languages. *Formal Aspects of Computing*, 22(3-4):297–326, 2010.

[12] D. Di Ruscio, F. Jouault, I. Kurtev, J. Bézivin, and A. Pierantonio. Extending AMMA for supporting dynamic semantics specifications of DSLs. Technical Report 06.02, Laboratoire d'Informatique de Nantes-Atlantique (LINA), Apr. 2006.

[13] F. Durán, F. Orejas, and S. Zschaler. Behaviour protection in modular rule-based system specifications. In N. Martí-Oliet and M. Palomino, editors, *Recent Trends in Algebraic Development Techniques (WADT 2012)*, volume 7841 of *Lecture Notes in Computer Science*, pages 24–49. Springer, 2013.

[14] F. Durán, S. Zschaler, and J. Troya. On the reusable specification of non-functional properties in DSLs. In *Proc. 5th Int'l Conf. on Software Language Engineering (SLE 2012)*, 2012.

[15] H. Ehrig. Introduction to the algebraic theory of graph grammars. In V. Claus, H. Ehrig, and G. Rozenberg, editors, *1st Graph Grammar Workshop*, volume 73 of *Lecture Notes in Computer Science*, pages 1–69. Springer, 1979.

[16] H. Ehrig, K. Ehrig, U. Prange, and G. Taentzer. *Fundamentals of Algebraic Graph Transformation*. Springer, 2005.

[17] H. Ehrig, G. Engels, H.-J. Kreowski, and G. Rozenberg, editors. *Handbook of Graph Grammars and Computing by Graph Transformation, Volumne II: Applications, Languages and Tools*. World Scientific, 1999.

[18] H. Ehrig, A. Habel, J. Padberg, and U. Prange. Adhesive high-level replacement categories and systems. In H. Ehrig, G. Engels, F. Parisi-Presicce, and G. Rozenberg, editors, *ICGT*, volume 3256 of *Lecture Notes in Computer Science*, pages 144–160. Springer, 2004.

[19] H. Ehrig and B. Mahr. *Fundamentals of Algebraic Specification 2. Module Specifications and Constraints*. Springer, 1990.

[20] H. Ehrig, U. Prange, and G. Taentzer. Fundamental theory for typed attributed graph transformation. In H. Ehrig, G. Engels, F. Parisi-Presicce, and G. Rozenberg, editors, *Graph Transformations, Second International Conference, ICGT 2004, Proceedings*, volume 3256 of *Lecture Notes in Computer Science*, pages 161–177. Springer, 2004.

[21] G. Engels, J. H. Hausmann, R. Heckel, and S. Sauer. Dynamic meta modeling: A graphical approach to the operational semantics of behavioral diagrams in UML. In *Proc. of the 3th International Conference on the Unified Modeling Language: Modeling Languages and Applications (UML'00)*, volume 1939 of *Lecture Notes in Computer Science*, pages 323–337. Springer, 2000.

[22] G. Engels, R. Heckel, and A. Cherchago. Flexible interconnection of graph transformation modules. In H.-J. Kreowski, U. Montanari, F. Orejas, G. Rozenberg, and G. Taentzer, editors, *Formal Methods in Software and Systems Modeling, Essays Dedicated to Hartmut Ehrig, on the Occasion of His 60th Birthday*, volume 3393 of *Lecture Notes in Computer Science*, pages 38–63. Springer, 2005.

[23] G. Engels, R. Heckel, G. Taentzer, and H. Ehrig. A combined reference model- and view-based approach to system specification. *International Journal of Software Engineering and Knowledge Engineering*, 7(4):457–477, 1997.

[24] M. D. D. Fabro, J. Bezivin, F. Jouault, E. Breton, and G. Gueltas. AMW: A Generic Model Weaver. *1ères Journées sur l'Ingénierie Dirigée par les Modèles*, 2005.

[25] T. Fischer, J. Niere, L. Torunski, and A. Zündorf. Story diagrams: A new graph rewrite language based on the unified modeling language. In G. Engels and G. Rozenberg, editors, *Proc. of the 6th International Workshop on Theory and Application of Graph Transformation (TAGT'98)*, volume 1764 of *Lecture Notes in Computer Science*, pages 296–309. Springer, 2000.

[26] U. Golas, A. Habel, and H. Ehrig. Multi-amalgamation of rules with application conditions in adhesive categories. *Mathematical Structures in Computer Science*, 24(4), 2014.

[27] U. Golas, L. Lambers, H. Ehrig, and F. Orejas. Attributed graph transformation with inheritance: Efficient conflict detection and local confluence analysis using abstract critical pairs. *Theoretical Computer Science*, 424:46–68, 2012.

[28] M. Große-Rhode, F. Parisi-Presicce, and M. Simeoni. Spatial and temporal refinement of typed graph transformation systems. In L. Brim, J. Gruska, and J. Zlatuska, editors, *Mathematical Foundations of Computer Science 1998, 23rd International Symposium, MFCS'98, Proceedings*, volume 1450 of *Lecture Notes in Computer Science*, pages 553–561. Springer, 1998.

[29] M. Große-Rhode, F. Parisi-Presicce, and M. Simeoni. Formal software specification with refinements and modules of typed graph transformation systems. *Journal of Computer and System Sciences*, 64(2):171–218, 2002.

[30] C. Guy, B. Combemale, S. Derrien, J. R. Steel, and J.-M. Jézéquel. On model subtyping. In A. Vallecillo, J.-P. Tolvanen, E. Kindler, H. Störrle, and D. Kolovos, editors, *Proc. 8th European Conf. on Modelling Foundations and Applications (ECMFA'12)*, volume 7349 of *LNCS*, pages 400–415. Springer, 2012.

[31] A. Habel and K.-H. Pennemann. Correctness of high-level transformation systems relative to nested conditions. *Mathematical Structures in Computer Science*, 19(2):245–296, 2009.

[32] R. Heckel and A. Cherchago. Structural and behavioural compatibility of graphical service specifications. *Journal of Logic and Algebraic Programming*, 70(1):15–33, 2007.

[33] R. Heckel, A. Corradini, H. Ehrig, and M. Löwe. Horizontal and vertical structuring of typed graph transformation systems. *Mathematical Structures in Computer Science*, 6(6):613–648, 1996.

[34] R. Heckel, G. Engels, H. Ehrig, and G. Taentzer. Classification and comparison of modularity concepts for graph transformation systems. In Ehrig et al. [17], chapter 17, pages 669–690.

[35] Z. Hemel, L. C. L. Kats, D. M. Groenewegen, and E. Visser. Code generation by model transformation: A case study in transformation modularity. *Software and Systems Modelling*, 9(3):375–402, June 2010.

[36] F. Jouault, F. Allilaire, J. Bézivin, and I. Kurtev. ATL: a model transformation tool. *Science of Computer Programming*, 72(1–2):31–39, 2008.

[37] J. Klein, L. Hélouët, and J.-M. Jézéquel. Semantic-based weaving of scenarios. In *Proc. 5th Int'l Conf. Aspect-Oriented Software Development (AOSD'06)*. ACM, 2006.

[38] J. Klein and J. Kienzle. Reusable aspect models. In *Aspect-Oriented Modeling Workshop at MODELS 2007*, 2007.

[39] H. Kreowski and S. Kuske. Graph transformation units and modules. In Ehrig et al. [17], chapter 15, pages 607–6380.

[40] S. Lack and P. Sobocinski. Adhesive categories. In I. Walukiewicz, editor, *FoSSaCS*, volume 2987 of *Lecture Notes in Computer Science*, pages 273–288. Springer, 2004.

[41] R. Machado, L. Foss, and L. Ribeiro. Aspects for graph grammars. *ECEASST*, 18, 2009.

[42] P. Manolios. *Mechanical Verification of Reactive Systems*. PhD thesis, University of Texas at Austin, 2001.

[43] J. Meseguer, M. Palomino, and N. Martí-Oliet. Algebraic simulations. *J. Log. Algebr. Program.*, 79(2):103–143, 2010.

[44] A. Moreno-Delgado, F. Durán, S. Zschaler, and J. Troya. Modular DSLs for flexible analysis: An e-Motions reimplementation of Palladio. In J. Cabot and J. Rubin, editors, *Proceedings 10th European Conference on Modelling Foundations and Applications (ECMFA 2014)*, volume 8569 of *Lecture Notes in Computer Science*, pages 132–147. Springer, 2014.

[45] OMG. Metaobject facility, 2014. `http://www.omg.org/mof/`.

[46] F. Parisi-Presicce. Transformations of graph grammars. In Cuny et al. [7], pages 428–442.

[47] J. E. Rivera, F. Durán, and A. Vallecillo. Formal specification and analysis of domain specific models using Maude. *Simulation*, 85(11-12):778–792, Nov. 2009.

[48] J. E. Rivera, F. Durán, and A. Vallecillo. A graphical approach for modeling time-dependent behavior of DSLs. In *IEEE Symposium on Visual Languages and Human-Centric Computing, VL/HCC 2009, Proceedings*, pages 51–55. IEEE, 2009.

[49] J. E. Rivera, F. Durán, and A. Vallecillo. On the behavioral semantics of real-time domain specific visual languages. In P. C. Ölveczky, editor, *Rewriting Logic and Its Applications - 8th International Workshop, WRLA 2010, Revised Selected Papers*, volume 6381 of *Lecture Notes in Computer Science*, pages 174–190. Springer, 2010.

[50] J. E. Rivera, E. Guerra, J. de Lara, and A. Vallecillo. Analyzing rule-based behavioral semantics of visual modeling languages with Maude. In *Proc. of the 1st Intl. Conf. on Software Language Engineering (SLE'08)*, volume 5452 of *Lecture Notes in Computer Science*, pages 54–73, 2008.

[51] L. M. Rose, E. Guerra, J. de Lara, A. Etien, D. S. Kolovos, and R. F. Paige. Genericity for model management operations. *Software and System Modeling*, 12(1):201–219, 2013.

[52] G. Rozenberg, editor. *Handbook of Graph Grammars and Computing by Graph Transformations, Volume I: Foundations*. World Scientific, 1997.

[53] D. C. Schmidt. Model-driven engineering. *IEEE Computer*, 39(2):25–31, Feb. 2006.

[54] A. Schürr, A. Winter, and A. Zündorf. The PROGRES-approach: Language and environment. In Ehrig et al. [17], chapter 13, pages 487–550.

[55] G. Taentzer and A. Schürr. DIEGO, another step towards a module concept for graph transformation systems. *Electronic Notes on Theoretical Computer Science*, 2:277–285, 1995.

[56] M. Tisi, F. Jouault, P. Fraternali, S. Ceri, and J. Bézivin. On the use of higher-order model transformations. In R. F. Paige, A. Hartman, and A. Rensink, editors, *Proc. 5th European Conf. on Model Driven Architecture – Foundations and Applications (ECMDA-FA'09)*, volume 5562 of *Lecture Notes in Computer Science*, pages 18–33. Springer, 2009.

[57] J. Troya, J. E. Rivera, and A. Vallecillo. Simulating domain specific visual models by observation. In *Proc. 2010 Spring Simulation Multiconference (SpringSim '10)*, pages 128:1–128:8. ACM, 2010.

[58] J. Troya, A. Vallecillo, F. Durán, and S. Zschaler. Model-driven performance analysis of rule-based domain specific visual models. *Information and Software Technology*, 55(1):88–110, 2013.

[59] A. van Deursen, P. Klint, and J. Visser. Domain-specific languages: An annotated bibliography. *SIGPLAN Not.*, 35(6):26–36, 2000.

[60] J. Whittle, P. Jayaraman, A. Elkhodary, A. Moreira, and J. Araújo. MATA: A unified approach for composing UML aspect models based on graph transformation. In S. Katz and H. Ossher, editors, *Transactions on Aspect-Oriented Development (TAOSD VI), Special Issue on Aspects and MDE*, volume 5560 of *LNCS*, pages 191–237. Springer, Oct. 2009.

[61] S. Zschaler. Formal specification of non-functional properties of component-based software systems: A semantic framework and some applications thereof. *Software and Systems Modelling (SoSyM)*, 9:161–201, Apr. 2009.

[62] S. Zschaler. Towards constraint-based model types: A generalised formal foundation for model genericity. In C. Atkinson, E. Burger, T. Goldschmidt, and R. Reussner, editors, *Proc. 2nd Workshop on View-Based, Aspect-Oriented and Orthographic Software Modelling (VAO'14)*, 2014.